University of London
Goldsmiths College
Department of Computing

TimeLines: Live Coding Music as Reactive Functions of Explicit Time

Dimitris Kyriakoudis
33413886
Supervised by Jamie Forth

Submitted in part fulfillment of the requirements for the degree of
Bachelor of Science in Music Computing of Goldsmiths
University of London, August 2018

Abstract

We consider music as a direct and explicit function of its domain, time. We explore how high-level musical and synthesis concepts, such as bars, beats, envelopes, and sequencers, can be modelled by equations that, given the current time, return the appropriate value. We propose a series of simple numerical functions, which can be composed to produce complex and arbitrary time-varying behaviours.

TimeLines, a live coding instrument for live performance and studio composition, is presented as an implementation of this approach. Informed by principles of Functional and Functional Reactive Programming, such as immutability, higher-order functions, and the explicit modelling of time and time-varying values, TimeLines enables the musician to externalise and manipulate creative thought processes using a clean, abstract syntax that is modular and scalable. In addition, it offers a customised text editing environment to maximise typing efficiency while reducing visual and manual distractions.

Overall, TimeLines aims to provide a responsive and evolving musical experience, shaped by the player's mental workflow and scaled by their mastery.

Acknowledgements

First and foremost, my unconditional gratitude belongs to my parents, who have never stopped supporting me in any and all ways they can, even though they neither understand nor care about what I do.

Secondly, special thanks are due to Jamie Forth, whose guidance and supervision not only made this project possible, but also influenced me as a musician programmer. Thanks for taking the extra time and effort to make sense of my ramblings, and thanks for showing me the light of SuperCollider and Emacs.

Lastly, my perpetual gratitude and deepest admiration to everyone involved in open source projects, dedicated to free and open software, and tirelessly helping people on chats and forums. Especially the people at the Haskell IRC channel, who are always keen to help and share their knowledge.

Contents

List of Figures

Chapter 1

Introduction

Humans experience music through their sense of sound, and they experience sound as oscillations of air pressure levels over time. These oscillations exist across space, for air is the medium in which they form and through which they propagate towards the human ear, but their experience exists across time, for time is the catalyst to their propagation and subsequent perception. The sequence and arrangement of these pressure levels in space, preceded and followed by a uniform stillness, contains the entirety of the information that, when perceived linearly over time, we experience as a full piece of music. From that information, the human brain tends to abstract high-level perceptual entities such as melodies, harmonies, and rhythms. These can be subsequently deconstructed to a slightly lower lever, that of individual notes, beats, and miscellaneous sound events. The next, even lower lever of auditory perception deals with the parameters of the sound wave itself over time, such as its amplitude, frequency, and timbre. Going any lower than that, one reaches the level of pure information, that of the instantaneous levels of air pressure for each and every moment of auditory experience. That is the level where human comprehension starts to fade, giving way to the clueless observing of the seemingly chaotic and meaningless stream of information we call music.

Interfaces for the expression and creation of music have, almost exclusively, been concerned with the higher conceptual levels of temporal organisation of sound. Loose, instantaneous sound is grouped into discrete notes and beats of well-defined duration, which are then grouped into clear and recurring temporal structures, such as the measure or the loop, in turn to be further grouped into phrases or sections and so forth. It is these structures that are then dealt with, composed, manipulated, and communicated by musicians, hiding away the layers of low-level information that are necessary for their experience. This conceptual grouping allows one to easily and predictably handle, manipulate, and reason about vast quantities of low-level data at once, but comes at the expense of relinquishing control over any aspect of sound and music that lies beyond those models.

Consider, instead, providing the musician with access to the lowest level that humans can comfortably comprehend, that of the instantaneous amplitudes and frequencies of waves and their over-time behaviour, yet still allowing them to build and use their own high-level temporal structures, which serve and encapsulate their unique understanding of music's organisation. One can then have the benefits of both high-level expressivity and low-level control. Using these custom temporal constructs, the musician can easily and intuitively organise and arrange the behaviour of the low-level data in time, from a piece's beginning till its end, with absolute precision where needed. To experience that piece, one would just have to traverse through that information from beginning to end, over an amount of time equal to its duration.

The contribution of this project is twofold. Firstly, it proposes an approach to thinking about and composing music as algebraic functions of explicit time. In mathematics, a function is a set of rules that matches every input from a domain to one, and only one, output. In the case of functions of time, we can think of the domain as a continuous range of real numbers, spanning from 0 to the duration of the piece, in seconds. For example, the time domain of a piece that has a duration of 3:30 minutes would be the inclusive range $[0, 3.5]$. For each input from that domain, a series of equations would dictate the values that each synthesis parameter should have at that moment in time, essentially answering the question "what should all my instruments be doing right now?" for every "now" in the piece.

The term "explicit" simply suggests that the flow of time is not modelled and dealt-with behind the scenes, as is usual with most music systems, but rather given a first-class presence and assigned a name for use anywhere in the parameter equations. In other words, time is treated as any ordinary number, which happens to constantly increase throughout a musical piece or performance. By adding to, multiplying with, or otherwise manipulating its value, one can define arbitrarily deep and complex musical structures, with a greater degree of flexibility and versatility than most other methods of reacting to the temporal flow.

Secondly, this project presents TimeLines, a live coding domain-specific language that implements the aforementioned method in a pure, Functional Reactive approach. The term "pure" refers to the fact that these functions have no computational side-effects; they are exclusively concerned with taking an input from the time domain and calculating the value each parameter should have at that time. No global state is changed, no objects are mutated, no envelopes are triggered, and no sequencers are clocked. All musical structure simply comes about by the linear progression of time and the changes that brings to the value of the parameters of fixed sound synthesis processes. In addition to the language, TimeLines provides a customised version of a text editor, designed around the needs of live coding and with typing efficiency and ergonomics in mind.

Beyond its implementation, TimeLines aims to propose a formalised methodology and practice for composing, producing, performing, and conceptualising of music as an explicit and direct function of the movement of time, both in the mathematical sense and otherwise.

1.1 Motivation

The desire for this project was born out of a personal frustration with the experience of the computer as an instrument for musical performance, composition, and production.

Most conventional music software is designed around Graphical User Interfaces (GUIs). This approach can prove problematic for a few reasons. Firstly, it offers little support for the algorithmic externalisation and representation of musical thought. Instead, the raw musical data is placed on a linear timeline, often manually, with no way of encapsulating the musician's thought process that resulted in that data in the first place. This has several consequences, such as the lack of support for manipulating this data in deep and meaningful to the composer ways. Instead of being able to change the parameters of the creative thought process that pieces the music together, one is forced to manually apply those changes by means of clicking and dragging on the raw material on the screen. This, in turn, makes it practically impossible to automate processes and workflows that may be described by even the simplest of algorithms, forcing the musician to spend more time manipulating data and less time developing musical thoughts. Lastly, these environments tend to adopt certain assumptions about the nature of music and its structure in time, making it harder for artists to develop their custom and unique conceptual frameworks and thus severely confining their creative output.

On the other hand, programmatic environments for the creation of music, provide a flexible and modular framework for artists to build their own understanding of music, their workflow, and ultimately their practice. Users can now abstract and automate processes that can be controlled with a handful of high-level parameter, to produce or manipulate vast amounts of musical material at a time. This allows one to slowly accumulate a selection of reusable algorithms that reflect their understanding of music and its internal and external structure, no matter how precise or arbitrary. Lastly, such environments allow musicians to explore previously uncharted territories of musical performance and composition, such as manipulating audio in extreme and novel ways, or using randomness, constrained or entirely chaotic, in real time to control any number of parameters.

These environments, however, come with their own shortcomings. In the case of visual languages, one's ability to quickly and effectively manipulate the code is limited by one's virtuosity of the computer mouse, which again leads to a lot of repetitive manual labour. On the other hand, textual languages can achieve much more with a few strokes of the keyboard, but the text manipulation capabilities of their accompanying editors are no different than those of a word document processor. Furthermore, a lot of those languages are based on high-level abstractions of various musical concepts, such as notes, scales, and patterns. While useful at times, the implementation specifics of these abstractions pose an additional learning challenge for users and limits their usage to what their developer envisioned and provided.

The combination of all of the above constitute the computer a highly capable instrument, but one that hardly enables or encourages artists to build mastery along with their personal workflow and practice. Between pointing and clicking at the screen, dragging notes and samples around a timeline, or reading the documentation for functions and classes, a musician is forced to consider the implementation details of their music more than the thought process itself that gives birth to the music in the first place.

1.2 Report Structure

2. This chapter sets the technical and creative context of the project. It examines different forms of musical notation and their representations of time. It introduces the paradigm of Functional Reactive Programming, exploring its ideas as a potential method for the notation and execution of musical content. It also introduces the practice of live coding in the context of music performance and composition.
3. This chapter presents a method for composing and developing musical material and structure through mathematical equations. It introduces a series of language-agnostic, multi-purpose control functions, as well as some of the possible ways to compose and use them.
4. This chapter describes the design of TimeLines, a live coding environment and purely functional implementation of the ideas developed in the previous chapter. All design choices, including the software utilised and how it was used, are discussed and justified, in accordance with the project's goals.
5. This chapter presents the implementation methods of TimeLines, both the code library and the editing environment. Special attention is paid to a few parts of their code, while their significance to the instrument's usability and flexibility is described.
6. This chapter evaluates the success by which TimeLines implements the concepts presented in chapters 1 and 2. It does so through a series of live performances and the resulting levels of flow that were experienced.
7. Lastly, this chapter provides a summary of the ideas developed in this project, as well as some pointers for future work and development.

Chapter 2

Literary, Technical, and Creative Context

The research context of TimeLines spreads its roots across multiple disciplines, from mathematical and psychological research into music, to the live coding practices of musicians and visualists alike.

2.1 Flow

To design and implement a digital music instrument that enables its player to build and maintain a state of mental flow, colloquially referred to as "the zone", the conditions which enable this state to be induced in the first place must be examined. First identified by Hungarian-American psychologist Mihaly Csikszentmihalyi, flow is described as a mental state of intense focus and concentration, which leads to a suppression of self-conscious reflection and complete immersion to an activity, even warping one's sense of time.[1]

This state of blissful concentration, experienced by expert musicians, athletes, and artists alike, comes as result of a clear interplay between challenge and skill. To avoid boredom and loss of interest, a task's difficulty should be able to scale to match one's skill and experience at the task, therefore constantly providing room for improvement. On the other hand, to avoid being overwhelmed by the immense challenge of a task, one should have a clear view of the end goal and a solid idea of how to achieve it. Immediate feedback is key for the evaluation of one's methods and progress, while at the same time all distractions from the activity should be minimised.

2.2 Music Notation and Time

Of an essential nature to systems developed for the notation of music, and indeed for music itself, is the representation and organisation of time. Therefore, a method for the notation of musical thought need not only specify the nature and implementation details of a desired musical event, but also its location and duration in time, both *relative* and *absolute* [2]. Different mediums for musical notation provide different frameworks for representing time and its effect on musical material.

2.2.1 Graphical

Such is the importance of time to music that most graphical forms of notation devote to it one of the only two dimensional axes available. From the arrangement of symbols in antiquity, the traditional western score, the punch-cards used in player pianos of the early 20th century, and through to the Digital Audio Workstations (DAWs) of today, variations of the same method of representing the timing of musical events have been employed: placing them on a linear timeline [3]. While these methods are perfectly suited for intuitively visualising the progression of data in time, they do little to express the creative thought process that leads to this progression.

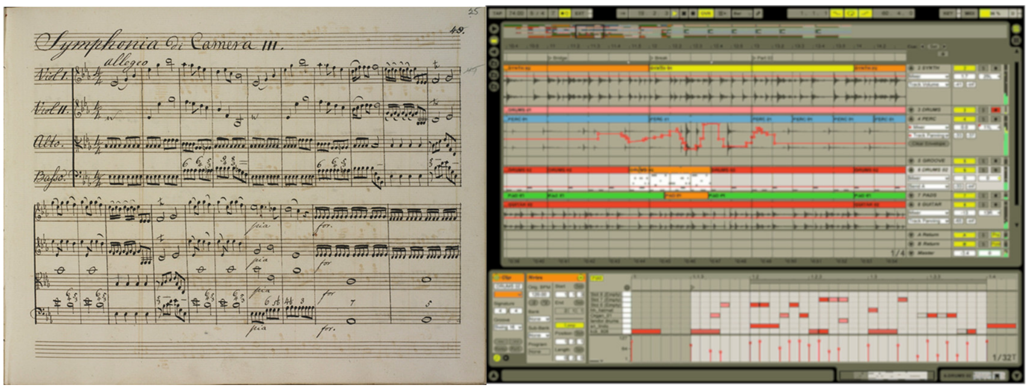


Figure 2.1: A 16th century score manuscript next to a typical modern DAW session. Image sources: publicdomainreview.org, futuremusic.com.

Other, freer and more abstract forms of graphical notation have been used by composers of the 20th and 21st centuries, to allow for a greater degree of the performer's interpretation to find its way into the music. These forms reject the traditional temporal linearity of notation, branching instead into using generative and algorithmic methods for the creation and representation of music.[3]

2.2.2 Mathematical

Mathematics provide a much more flexible and powerful form of notating musical parameters and their behaviour over time. Their ability to relate values to one another using simple, modular, and reliably composable rules allows for notating the relationships between all parameters that make up a piece of music, including time itself. This is achieved in terms of functions, which map values from one domain to their counterparts in another and can be arbitrarily combined to express even the most complex of behaviours between two or more parameters. In the case of timing, various functions can be used to represent the relationship between various notions of time, such as tempo functions, time-shifts, and time-maps.[2, 4]

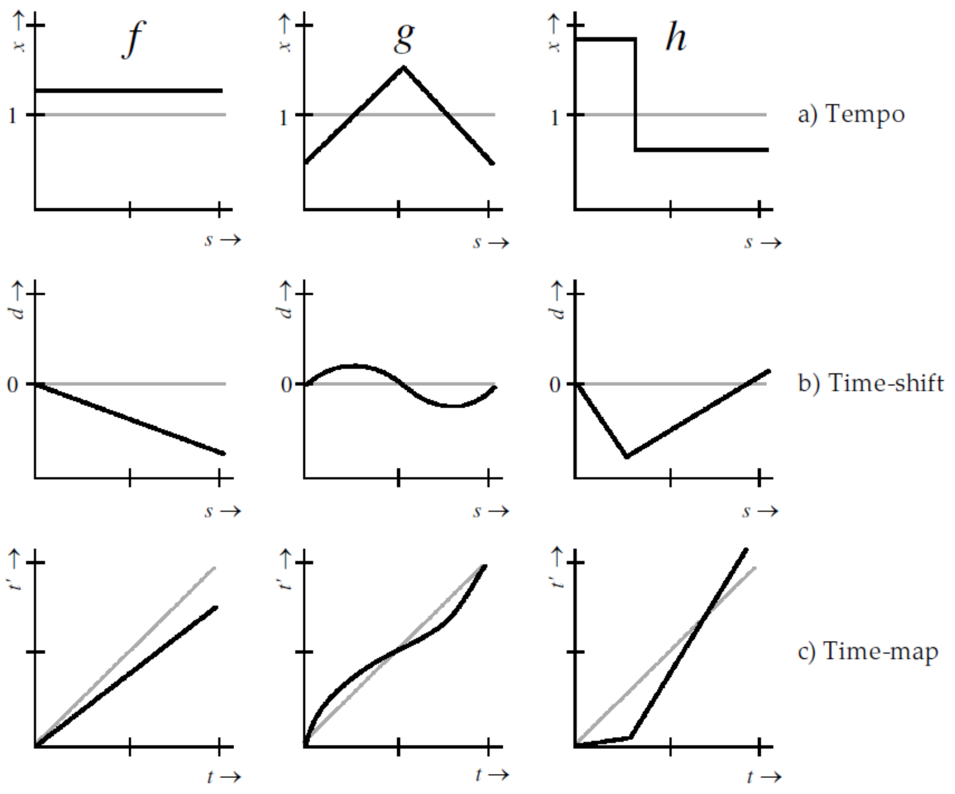


Figure 2.2: Representation differences between tempo functions, time-shifts, and time-maps. Figure source: Henkjan Honing

The latter, time-maps (also called time *deformations* or *warps*), are of special interest to this project, as they allow one to express warpings of one temporal domain to another, such as from the absolute time of a listener to the relative time of the performer's score. They are, essentially, numerical functions that, given the current time in one temporal domain and context, will return the time in another.

2.2.3 Algorithmic

Lastly, the broader and arguably most flexible method of notating musical material and thoughts is using algorithms. While a relatively vague term, it generally involves the direct expression of a thought process itself, not its outcome. When writing music as algorithms, the artist simultaneously creates and utilises a scaffolding for algorithmic thought, unloading cognitive load from the brain onto the code and therefore enabling further exploration from a new perspective.[3]

Algorithmic notations can support the development of experiential flow by providing simple primitive operations, which can be combined ad infinitum to form larger and ever more complex structures [5]. In contrast to visual notations, which aim to help the user recognise tools and their purpose, algorithmic code relies on the user memorising and recalling the solution to a problem, thus gradually building mastery of the language and its usage, akin to the virtuosity of an instrumentalist that has memorised all possible chords and finger positions.

The use of algorithmic code as notation has largely been enabled by the practice of live coding, whereby algorithms are written and manipulated while they are being executed, thus being "brought to life" and allowed to be interacted with.

2.3 Functional Programming

The Functional Programming (FP) paradigm has been steadily rising as a counterexample to the dominant, imperative style of Object Oriented Programming (OOP). Whereas OOP regards programs as sequences of commands dictating the manipulation of the internal and global state of data structures stored in memory, the functional approach thinks of computer programs as long chains of "pure" function applications to arguments, starting from the program's input arguments and continuing all the way through until the program's final output is returned. In a functional language, the imperative approach of telling the computer *how* to manipulate data to achieve an outcome is abandoned in favour of a more declarative semantic model, which instead describes to the computer *what* that result actually is.

John Backus, in his 1977 ACM Turing Award Lecture [6], states:

> Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor–the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Functional programming's answers to those critiques come, mostly, in the form of three main features [7]:

1. *Immutability.* Data structures are no longer mutated in place, but instead propagated through a series of sideeffect-free functions that return modified versions. Different parts of the program can now safely operate on these data structures in parallel, without fear of other threads interfering with them.
2. *Laziness.* Data will no longer be calculated as soon as it is defined, but rather when it is actually needed to complete a computation. Programmers can now work with infinite data structures and algorithms.
3. Higher-order functions. Functions will no longer be treated separately to the data they operate upon, but will instead be themselves considered data and used to compose larger,

more powerful and complex functions.

While all three are of great use to a musician programmer, it is the latter that marks the most dramatic change in usability:

> To be able to perform multiple transformations (as required by music sequencers and expression editors), or compose a number of transformations into a complex transformation (essential in programming languages for music or in combining partial computational models of expressive timing), the transformations themselves should be an object within the representation, not just functions applied to it.

What Honing is arguing for here [2] is essentially higher order functions: the representation of manipulations as ultimately another form of data, one which can be added together and composed much in the same way that the data they operate upon can.

2.3.1 Functional Reactive Programming

Of particular interest to this project is a certain subset of Functional Programming, that of the Functional Reactive Programming (FRP) paradigm. Originally proposed in Conal Elliott's and Paul Hudak's "Fran" animation library in 1997 [8], FRP is a computational paradigm based on two core principles: the explicit modelling of time and the declarative combining of time-varying values. Variables, commonly called "behaviours" or "signals", are assumed to be time-dependent and their value is expected to change at any moment during the course of a program's execution, without the programmer's intervention. Thus, the semantics of operating with such variables are not concerned with their current, or instantaneous, value, as is the case with imperative programming, but rather with their respective value for each moment in time.

As a simple example, consider the expression $a = b + c$. In an imperative language, this expression would be interpreted as an assignment operation and executing it would assign the value of the variable a to be equal to the sum of the values that b and c hold at the moment of execution. If, at any moment later in time, the values of either b or c were to change, then for the equality statement to hold true again the assignment expression would have to be re-evaluated.

On the contrary, in a Functional Reactive language, $a = b + c$ is not an operation of value assignment, which simply sums the two values and stores the result at a location in memory referred to by the variable a. Instead, it is a declaration of symbolic equality, a mathematical relation binding the three symbols, which, in the spirit of FP, is immutable and therefore will always hold true until rewritten. The programmer can now regard the value of the symbol a to be equal to the sum of whatever values b and c hold, at any moment, forever.

This has serious implications for the programmer constantly working with time-varying values, such as in the case of music or animation. One can now operate on material at much higher semantic level than before, concerned with the ways that material is related and composed in time rather than the specifics of the implementation of their relations and compositions.

2.4 Live Coding

Various definitions have been suggested for live coding, each of which is informed by a subset of the plethora of languages and environments available, as well as their resulting workflows. Some of those definitions focus on the notion of writing and modifying a computer program while it is running, while others further emphasise the performative aspect of the activity by describing how "performers create time-based works by programming them while these same works are being executed".[9]

While assigning a single umbrella definition to all live coding practices may be unwise, or indeed impossible, there are two characteristics that underlie practically all live coding activities: the use of computer code as the main, first-class medium for producing work, and the significance, if not necessity, of the interactive feedback loop for the development of said code. In other words, the artist doesn't simply use code as part of the activity, the writing and modifying of code *is* the main part of the activity itself. The artist is therefore no longer *just* an artist that uses code, but an artist-programmer hybrid, the *programmer artist*.[10]

We can distinguish between two major kinds of live coding: online and offline activities. An online activity is one whose creative output is rooted in the same temporal domain as the development of its code. In other words, when performing live in front of an audience, a musician's iterative interaction with the code is inherently synchronous with the musical outcome perceived by the audience. On the other hand, during an offline live coding session, the programmer's interaction with code is the process that leads to the resulting artwork, not an inherent part of it. The writing of code is not the artwork itself, but rather the canvas and brush that the artwork is "painted" with.

2.4.1 TidalCycles

Based in the functional language Haskell and a Functional Reactive approach to performing with time, TidalCycles (Tidal for short) utilises its host language's syntactic terseness and higher-order functions to create an abstract representation of time-based patterns, which provides a uniform framework for combining simple primitive patterns into complex time-evolving structures, which can be used to control any and all kinds of musical parameters.[11]

The principal contribution of Tidal to this project is the utilisation of Haskell's type system and syntax to provide the user with a clean, modular framework for combining and sequencing abstract representations of time-varying patterns.

2.4.2 Sonic Pi

Sonic Pi's main influence on this project is its use of a custom text editor, which provides an environment designed specifically for the use of live coding and incorporating features such as extensive keyboard shortcuts, for controlling both the text and audio engine, an integrated documentation and tutorial system, and a real-time visualisation of the produced audio.

2.4.3 Extempore

Extempore follows the approach of *temporal recursion*, whereby defined musical functions are made to schedule themselves in the future. Between callbacks, the user can change the function in dramatic ways, hearing the results of those changes the next time the function evaluates itself. While this approach requires a large amount of boilerplate code to be written in order to create a new function from scratch in real time, heavy use of snippets can streamline the process and instead allow the user to focus on the behaviour of the function, rather than its implementation.[12]

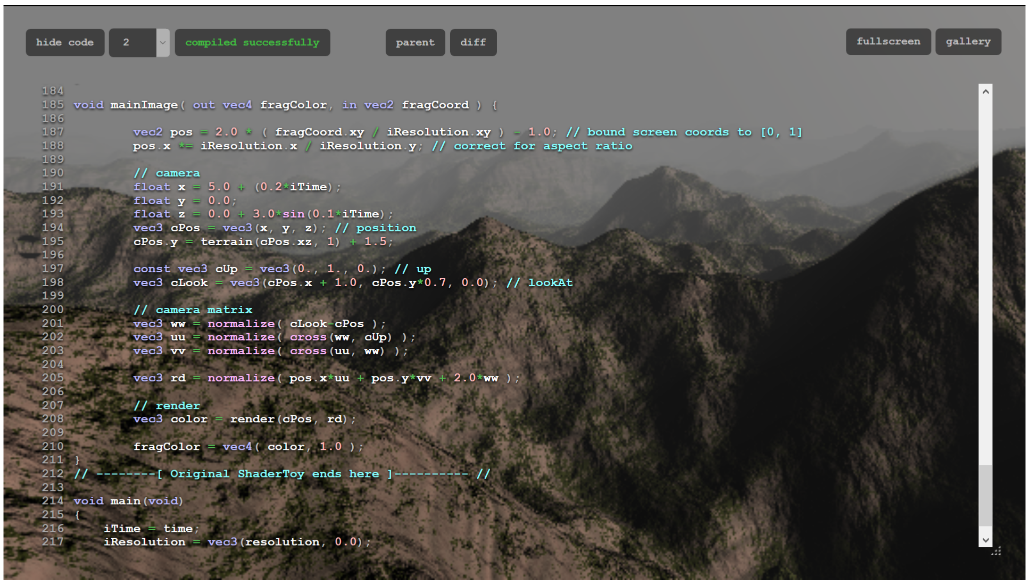
Extempore's contribution to this project is the automation of scaffolding required by the chosen workflow, by means of preconfigured snippets that serve multiple purposes.

2.4.4 Shaders and the Demoscene

The Demoscene is an internationally developed subculture that focuses on creating audiovisual computer art using code. While not all demos have to be live coded, quick and responsive feedback loops of interaction with the code are so important that a plethora of live coding tools have been developed, some to be kept for private use within teams and plenty to be freely shared on the internet. Some of these tools, such as GLSLSandbox, overlay the code on top of the produced image, while the user need not even explicitly evaluate changes to the code; the code gets automatically re-compiled whenever it has changed and contains no errors.

Events, called "Demoparties", are organised where teams will be given some time, typically one or two days, to live code a demo from scratch and finally present their work on the closing day. These events are a perfect example of an offline coding practice: the resulting artwork is not unfolded while the creators are engaged in the live coding, but rather when the code is finished and the team is ready to press "play" and share their work with the rest of the audience. That is not to say that the importance of coding live is diminished. On the contrary, during these events competitions are held where contestants will live code *fragment shaders*, programs that run in parallel on GPUs for every pixel, while having their screens projected for the audience.

Demo programming is oftentimes functional and reactive in nature. While state manipulation is still used in order to achieve optimal hardware performance, the sequencing of all the scenes is achieved solely by reacting to the immutable input of passing time. In other words, the program



```glsl

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {

        vec2 pos = 2.0 * ( fragCoord.xy / iResolution.xy ) - 1.0; // bound screen coords to [0, 1]
        pos.x *= iResolution.x / iResolution.y; // correct for aspect ratio

        // camera
        float x = 5.0 + (0.2*iTime);
        float y = 0.0;
        float z = 0.0 + 3.0*sin(0.1*iTime);
        vec3 cPos = vec3(x, y, z); // position
        cPos.y = terrain(cPos.xz, 1) + 1.5;

        const vec3 cUp = vec3(0., 1., 0.); // up
        vec3 cLook = vec3(cPos.x + 1.0, cPos.y*0.7, 0.0); // lookAt

        // camera matrix
        vec3 ww = normalize( cLook-cPos );
        vec3 uu = normalize( cross(ww, cUp) );
        vec3 vv = normalize( cross(uu, ww) );

        vec3 rd = normalize( pos.x*uu + pos.y*vv + 2.0*ww );

        // render
        vec3 color = render(cPos, rd);

        fragColor = vec4( color, 1.0 );
}
// --------[ Original ShaderToy ends here ]---------- //

void main(void)
{
    iTime = time;
    iResolution = vec3(resolution, 0.0);
```

Figure 2.3: The browser-based shader live coding environment GLSLSandbox.

structure of a demo can be thought of as "time goes in, demo comes out" [13]. They are not only reactive to time, but also space: since fragment shaders run in parallel for every pixel, all visuals are created as an explicit function of pixel coordinate, using mathematical equations that get abstracted into concepts like scenes and objects. Therefore, the only two arguments required to calculate the colour of a pixel are time and that pixel's x and y coordinates.

This mathematical approach to creating arbitrary content as a function of its domain became the main inspiration behind TimeLines.

Chapter 3

Music as a Function of Explicit Time

For any given moment in the duration of a musical piece or performance, there can only be a single value associated with every parameter of every process that constitutes it. As an example, consider the simple case of performing with a monophonic synthesizer. All possible sounds that a synthesizer can produce are a direct result of the specific configuration of all its parameters, which may include oscillator amplitude and frequency, filter cutoff and resonance, amount of waveshaping and effects, etc.

Therefore, any possible performance with this synthesizer can be conceived of as a set of sequences of particular values, one for each of its parameters, unfolding in parallel. In order to play a melody, a performer combines increases and decreases in oscillator amplitude, usually achieved by triggering an envelope with a keyboard, with discrete changes in the oscillator's frequency, similarly controlled by use of the keyboard and thus synchronised with the triggers of the amplitude envelope. Therefore, if we were to assign sliders to the instantaneous amplitude and frequency of an oscillator, we could perform any conceivable melody, with any possible phrasing, by very precisely manipulating those sliders in time, circumventing the need for keyboards and envelopes all together. In the case of a polyphonic synthesizer of, say, 8 or 16 voices, the same could be achieved by having a pair of sliders for each oscillator's amplitude and frequency. Then, any polyphonic melody, chord, arpeggio, or otherwise polyphonic piece of music can be performed by carefully and precisely moving these sliders by the right amounts and at the right moments.

The instrument used to illustrate this example, the electronic synthesizer, was chosen for a reason: it is a well-defined sound process with a fixed, if rather small, number of parameters, whose values affect its sonic output in predictable, reproducible, and easily conceivable ways. Acoustic, electric, or otherwise mechanical instruments on the other hand pose a much harder challenge for anyone looking to abstract them into a handful of meaningful parameters that can be easily managed. Still, precise physical models for these instruments could provide just enough abstraction to be able to isolate parameters of a high enough level, such as lip tension and placement for wind instruments, pluck or bowing force and positioning for stringed instruments, or hit strength and location for a percussive body.

To generalise, the total number of values v_{total} that has to be calculated for a piece of duration T sampled at a rate of f_s, which consists of n distinct sound processes, each with its own p_i number of parameters, can be derived by:

$$v_{total} = T f_s \sum_{i=0}^{n} p_i \tag{3.1}$$

The rate of observation f_s will determine the upper limit of frequencies f_l that can be represented to be $f_l = f_s/2$, as per the Nyquist-Shannon sampling theorem.

3.1 Basic Functions and their Combinations in Time

The immediately obvious benefit of using mathematics as a means of expressing and creating these sequences of values is their complete modularity and composability. Operations can be combined, sequenced, and nested in arbitrary amounts and ways, thus providing a highly flexible framework for the expression of even the most complex of musical thoughts. This ability to gradually build up complexity by combining simple, reproducible, and transparent parts allows one to start simple and gradually build their skill and understanding, moving to higher complexity once mastery of the previous level has been achieved. This, in turn, provides a constantly challenging yet clearly rewarding framework for a musician to develop their musical skills and understanding, as well as formulate their own practice and methodology.

The following is a list of simple functions, together with a description of their implementation and possible usage:

- *Addition and Subtraction.* These simple operations, when applied to values, can be thought of as behaviour combinators. Simply summing two signals results in a third, which exhibits properties of the behaviour of both. When applied to its time reading, addition and subtraction can shift the temporality of an event or structure around, moving it backwards and forwards respectively.
- *Multiplication and Division.* Provide the ability to stretch and compress the behaviour of signals, both in the temporal and value domains. Special meaning can be attributed to multiplication with 1 and 0, which can be interpreted as switching a signal on and off respectively. When applied to time, multiplication and division can be thought of as speeding up and slowing down respectively.
- *Exponentiation.* When applied to normalised signals, exponents can be used to curve a signal's distribution, with exponents larger than 1 skewing values down (*acceleration*) and exponents less than 1 skewing values up (*deceleration*).

- *Step: (if time <= step then 0 else 1).* Can be used to represent a change of state after a certain point in time, while its behaviour can be inverted by being subtracted from 1.
- *Switch: (if start <= time => end then 1 else 0).* Can be used to switch signals or behaviours on between two arbitrary moments in time, essentially conducting the entrance and exit of various signals. Like above, its behaviour can be inverted by subtracting it from 1.
- *fromList*: `list[floor(listLength * normalizedPhasor)]`. Can be used to introduce arbitrary sequences of numbers or signals by indexing through a list using a phasor normalised between 0 and 1, or even other functions like sine waves. The indexing signal's frequency determines the frequency of the subsequent data sequence.
- *Modulo.* By applying modular arithmetics to signals, linear behaviours and structures can be made cyclical and repeating in nature. When combined with multiplication, a phasor can be made to repeat a given number of times over its original duration.
- *Random*: extracting the absolute fractional part of a scaled or frequency-modulated sine wave. Pseudo-random algorithms can be created by warping the output of a regular function, such as a sine wave, to the point where its pattern is no longer recognisable. Indexing through that randomness can be achieved by discretising scaled phasors, where the amount of scaling determines the frequency by which random numbers are sampled.

By combining these functions in various ways, compound functions can be made to achieve almost any kind of modulation desirable. The following figures demonstrate a few examples of the temporal structures that can be achieved, both traditional and novel.

Figure 3.1 demonstrates modelling a typical 4/4 bar structure. By combining scale and modulo operations, various phasors can be derived from the linear ascent of time to provide discrete and continuous indexes of the current bar, current beat in the bar, and percentage of every beat's duration.

In figure 3.2, the behaviour of a typical Attack-Decay envelope is modelled, with the ability to specify the attack and decay times in seconds, as well as curve parameters for the two stages. The implementation of such a function scales and offsets time to linearly interpolate from 0 to 1 and back to 0 again, applying the curve parameters as exponents.

```
env atk rel c1 c2 t
  | t > atk + rel = 0
  | t < atk = (t/atk)**c1
  | otherwise = (1 - (t-atk)/rel)**c2
```

Code Snippet 3.1: Example implementation of an AD envelope with separate curve parameters.

Figure 3.3 demonstrates how arbitrary values can be sequenced using `fromList` and used to modulate parameters of other signals, such as a sine wave's frequency. Notice how the sine wave's phase is manually driven by a phasor, whose frequency can also be used to modulate the resulting wave.

Lastly, figure 3.4 demonstrates more complex and experimental behaviour for signals. It utilises multiplication with 0 to nullify the output of signals for certain moments in time, thus achieving the illusion of sequential behaviour when these signals are summed together.

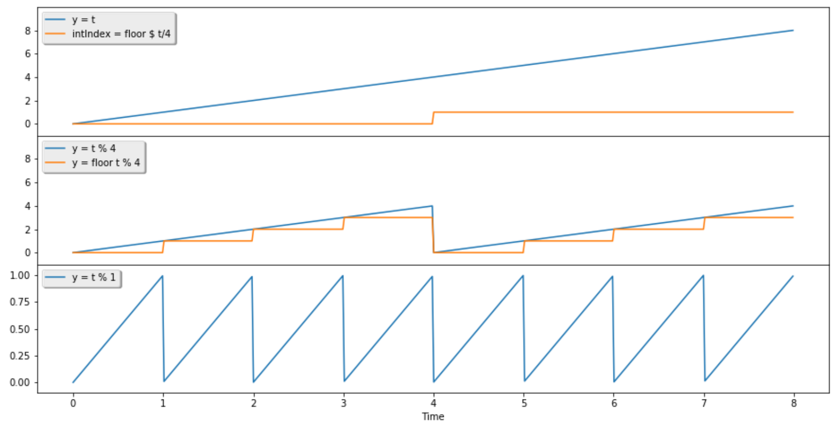


Figure 3.1: A traditional 4/4 bar structure.

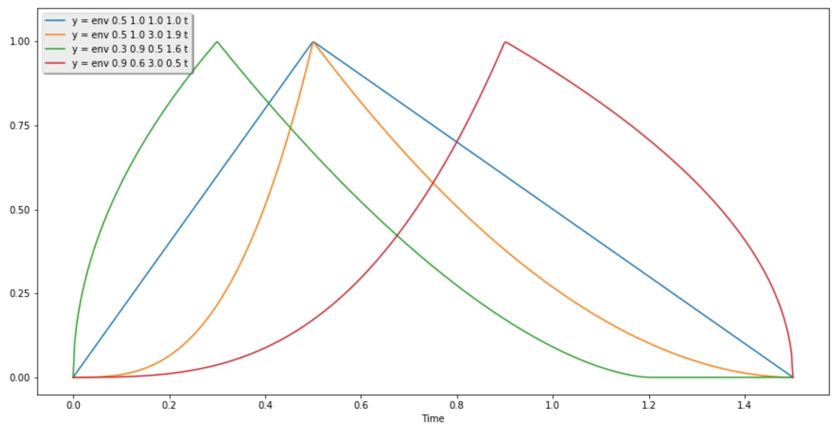


Figure 3.2: Attack-Decay envelopes that operate on absolute time with different curvatures.

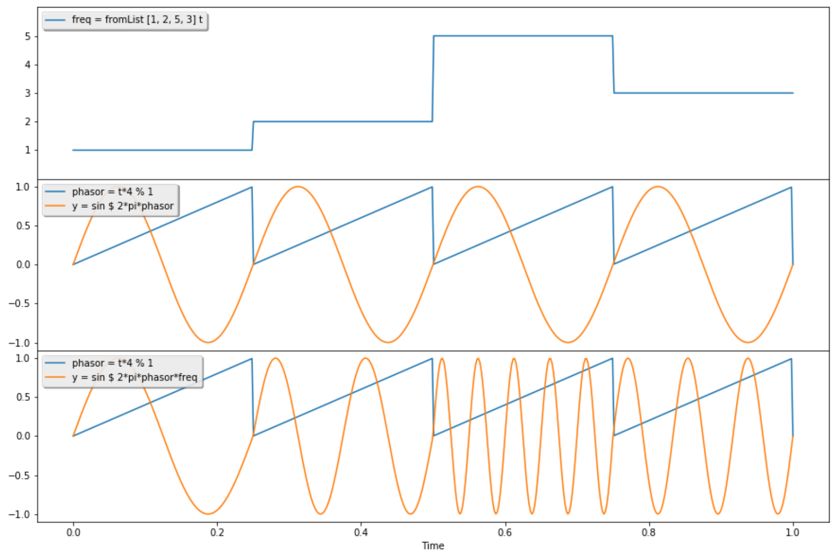


Figure 3.3: Using an arbitrary sequence of numbers to modulate a sine wave's frequency.

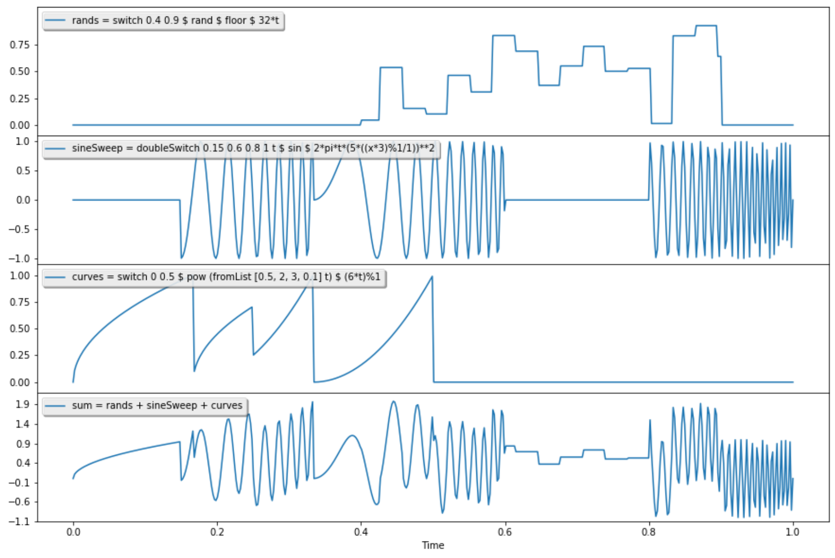


Figure 3.4: Switching signals on and off over time and summing them to achieve compound and arbitrarily complex behaviour.

3.2 Mathematics as Notation

We saw that using algebra as the underlying language for musical content creation allows for extreme power and flexibility using minimal, almost universal syntax and a high degree of composability which scales with the musician's needs, skills, and experience. Instead of having to learn about a variety of different classes and their methods, one can reuse the same simple mathematical principles, concepts, and functions for every aspect and parameter of music. What's more, it does so through a uniform framework using simple, transparent, and reproducible rules and syntax, which are approachable for the beginner and capable of scaling for the master.

Chapter 4

TimeLines: Design

Time is central to music, and presents a few technical hurdles to jump. If you have more than one music-generating script running at the same time, you'll want to keep them in sync somehow, and if you're playing with someone else, you'll need to keep in sync with them as well. You not only need to make sure all the different scripts are playing at the same speed, but also in phase - that is, everything needs to be able to work out when the start of the bar is. Further, at some point you'll want to change the speed of the music, especially if your crowd are looking a bit restless. And if you want to start up a new script while another is running, how do get it to start at the right moment?[14]

In the previous Chapter we saw how any monophonic or polyphonic piece of music could theoretically be performed by very accurately and precisely manipulating the right amount of sliders in time. Of course, this is practically impossible to achieve, especially within the confines of real-time performance, and even more so when there is only one performer available. One could, however, model the future flow of time beforehand and systematically compute all the necessary values in advance. By employing the arbitrary precision of mathematics and the computational power of modern computers, the low-level data of minutes worth of music can be precisely and efficiently calculated in just seconds, and stored for future access. Then, "moving the sliders in time" simply becomes "indexing through the data at the right rate", with the resulting information streams, and therefore music, in both cases being identical for the listener.

This approach is no different to what music software has been doing all along: precomputing future values in the form of audio buffers, which are sent to the Digital-to-Analog Converter (DAC) and output to the speakers while the next batch of audio is being calculated. With software that should react to incoming input in a timely manner, such as virtual instruments played by controllers or effects processes operating on real-time audio, then the length of the pre-calculated buffer has to remain as short as computationally possible, so as to reduce the latency of reaction to any incoming information. A live coding scenario, on the other hand, can be imagined as a closed system: a coded stream of information need not respond to anything outside of the code itself. Once the code is changed by the performer, the data calculated from it change too. If the new data is pre-calculated for the same period of time as before, then their indexing continue normally, from the exact same position in the buffer relative to its beginning as the previous data left off, essentially applying the code changes to the music instantly and assuring tempo and phase synchronicity for free.

TimeLines differs fundamentally from many other live coding systems by design, in that there are no real-time, continuously running processes which are modified by applying changes to the code. The only running processes are the indexing of buffers and the subsequent sound synthesis based on those values, but they are mostly left untouched during a session, save for their instantiation and freeing. Instead, it is the indexed data itself that is live coded, and thus all resulting music comes about as a result of the different ways that data is put together.

4.1 Software Used

The first stage of the design process was the selection of software tools and platforms that were to be used. Not only would this greatly affect the efficiency and effectiveness of the development process, but it would also highly influence the functionality of the resulting software.

The following criteria, informed by the overall goals and aims of the project, were used to guide the choice of software that was used to put together TimeLines:

1. Accessibility and openness. All software used be free, open-source and easily accessible by anyone, regardless of their platform or operating system of choice.
2. Portability. Besides being able to run on most architectures and operating systems, the software should be easily configured by portable and reproducible means (i.e. a cross-platform configuration file that can be re-used on a fresh installation).
3. Modularity, extensibility, and customisability. The ability to modify, extend, or otherwise rewire all pieces of software that make TimeLines is essential, not only for the purposes of its development but also for its growth as a personal musical instrument.
4. Efficiency. To ensure the best possible interactive and computational performance, as well as to support less powerful systems like older hardware and the Raspberry Pi, the software used should be optimised for performance and able to be stripped of unnecessary and distracting features.
5. Maturity and stability. To avoid wasting time on technical issues not directly related to the development of this project, the software used should have reached a certain level of maturity and stability, with plenty of resources available for troubleshooting.

6. Strong user communities. Not only will this aid in the development of TimeLines by providing a plethora of available extensions and support from peers, it will also provide a fertile ground for the exchange of ideas and the development of both the project and its users.

All software candidates were evaluated with the above criteria in mind and the following were determined to be the right choice for this project.

4.1.1 SuperCollider

The first and easiest choice of software was that of the sound synthesis engine itself, SuperCollider. Originally released in 1996 by James McCartney and open-sourced in 2002, SuperCollider is now one of the most mature and most capable audio engines. The software is split into two distinct, yet tightly cooperating, parts: the audio back-end server (SCSynth) and a high-level, dynamically-type, object-oriented language that acts as a user interface to it (SCLang). Below is a closer look at the features of these two parts that make them suitable for this project, as well as how these features were used:

1. **SCSynth**: Two of SCSynth's most appealing features are its complete modularity and flexibility, as well as extreme computational efficiency. The fundamental building block of audio processes are *UGens*, short for Unit Generators. UGens are objects written in C and serve a specific purpose, such as synthesising, filtering, or otherwise modifying signals. Most UGens have been optimised for performance and can operate at either *Audio Rate* (matching the server's sampling rate, typically $44.1kHz$) or *Control Rate* (typically $700kHz$, can be changed), with many supporting either option. A combination of any number of UGens can be put together into a *SynthDef*, a blueprint outlining the structure of a *synth*, instances of which can be created and removed dynamically from the server's ordered execution node tree. Each synth, or node, can input from and output to any number of *buses*, each of which can either be running at audio or control rate. The input and output routing of each node can be dynamically changed at runtime, much like how a modular synthesizer's patch cables allow for the real-time rerouting of its signal flow. Nodes can then act as sound sources, synthesising or reading pre-recorded audio, as sound processors, reading and modifying the output of another node, or both. Therefore, SCSynth provides a highly flexible and capable playground for sonic experimentation, one that leaves few features to be desired by a live coder and that can efficiently run on all three major operating systems (Linux, OS X, Windows) and less capable hardware, such as older computers,inexpensive laptops, and Raspberry Pis. Originally, the server was integrated with the client language. Lastly, the server is solely controlled via OSC (Open Sound Control) messages, allowing any OSC-capable software to have complete control of its state and behaviour [15]. This has made SCSynth extremely popular among live coding projects, with many (e.g. TidalCycles, Sonic Pi, FoxDot, Overtone) utilising

the server's sonic potential through different languages (Haskell, Ruby, Python, and Clojure respectively) that better suit their design and creative philosophies.

Lastly, the popularity of SuperCollider brings a plethora of learning resources available for beginners and experts alike, as well as numerous server extensions and custom UGens, with guides for writing one's own.

2. **SCLang**: The language bundled with SuperCollider is an object-oriented, dynamically typed, high-level language with a C-like syntax, that draws its inspiration from Smalltalk and various features of functional programming (such as mapping and filtering collections). SCLang was designed specifically for the purposes of SuperCollider and as such comes equipped with many abstractions that help organise the server's many features, as well as ways to monitor and keep track of its state. Such abstractions include classes that function as language-side representations of server-side entities, such as UGens, synths, buses, buffers, groups etc. This makes it an ideal candidate to keep track of the server's state and make sure entities are created and freed where and when appropriate.

4.1.2 Haskell

Since Functional Programming was the focus of the project from the start, Haskell naturally was one of the first languages to come up during the research stage. It is a strongly and statically typed, lazily evaluated language that encourages, almost enforces, the Functional paradigm by placing an emphasis on the purity of functions. Unless explicitly stated otherwise, functions are assumed to take inputs and produce reproducible outputs, thus severely limiting the parts of the program that could cause state-related bugs.

Apart from all of Functional Programming's characteristic features, such as lazy evaluation and higher order functions, what sets Haskell apart, and the most obvious to the user, is its syntactical simplicity and elegance. In contrast to most other languages, Haskell manages to make its code aesthetically pleasing and more easily readable by simplifying the most commonly used operation: function application. Instead of using parentheses to delimit the beginning and end of arguments, a function is applied by means of a mere white space between the function's name and its arguments. Other syntactic conveniences, such as the dollar sign assigning precedence to everything that comes to its right, make for a minimal syntax that is easy to read, write, and reason about, even in the tight time constraints of live performance.

4.1.3 Emacs

The last piece of software to be decided upon, after the audio server and the interface language, was the text editor that would join the two and provide the interactivity required for live coding. After reviewing the available editors with the aforementioned design criteria in mind, it came down to the two that seemed to offer the best combination of popularity and extensibility: Atom and Emacs. Both of these editors have highly active communities and a plethora of extension packages available, with the ability to write one's own using powerful general-purpose programming languages (CoffeeScript and Emacs-Lisp respectively). Despite Atom's more visually appealing and beginner-friendly interface, compared to Emac's rather disheartening out-of-the-box experience, Emacs proved to be the more suitable of the two for a few reasons.

The development of Emacs started in the mid 70s and actively continues to date, marking it as one of the oldest and longest standing software projects. As such, it has achieved an extreme level of maturity, stability, and performance, in contrast with Atom's relatively recent development and prioritisation of UI over CPU and memory performance.

Emacs' longevity it has resulted in a strong and dedicated community of users and developers, who more often than not fall under both categories for one simple reason: Emacs can be better described as a text editor built with and around an Emacs-Lisp interpreter. Besides a small core of functionality written in C, the rest of the editor's feature set is written in a language that can be interpreted at run-time, essentially allowing the user to live code the editor itself, using it to change and augment its own functionality. This encourages the user to experiment with customising and extending their editor from early on, making for a constantly evolving relationship with one's software tools and a user-tailored workflow and experience. This is exactly the sort of relationship TimeLines is aiming to cultivate with its users, thus making Emacs the obvious choice for a host interaction environment.

4.2 Structure Overview

In order to avoid having to constantly evaluate the equations and update the synths on the server with the resulting values, which is a time-sensitive process and prone to glitches and delays, the aforementioned approach of a pre-computed buffer was favoured for the design of TimeLines. Instead of calculating milliseconds worth of audio at audio-rate sampling rates, as music software typically does, TimeLines samples seconds or even minutes worth of control signals, at much lower rates, that guide the real-time synthesis of that audio. These are saved into audio-format files of relatively low sampling rate, which are then loaded by the server and indexed at the appropriate rate. A diagram outlining the internal workflow of TimeLines is described by figure 4.1.

By always evaluating these signals over the same start and end time points, one is assured that buffers can be hot-swapped while playback is continuing and that the new audio will be synced to and in phase with the old.

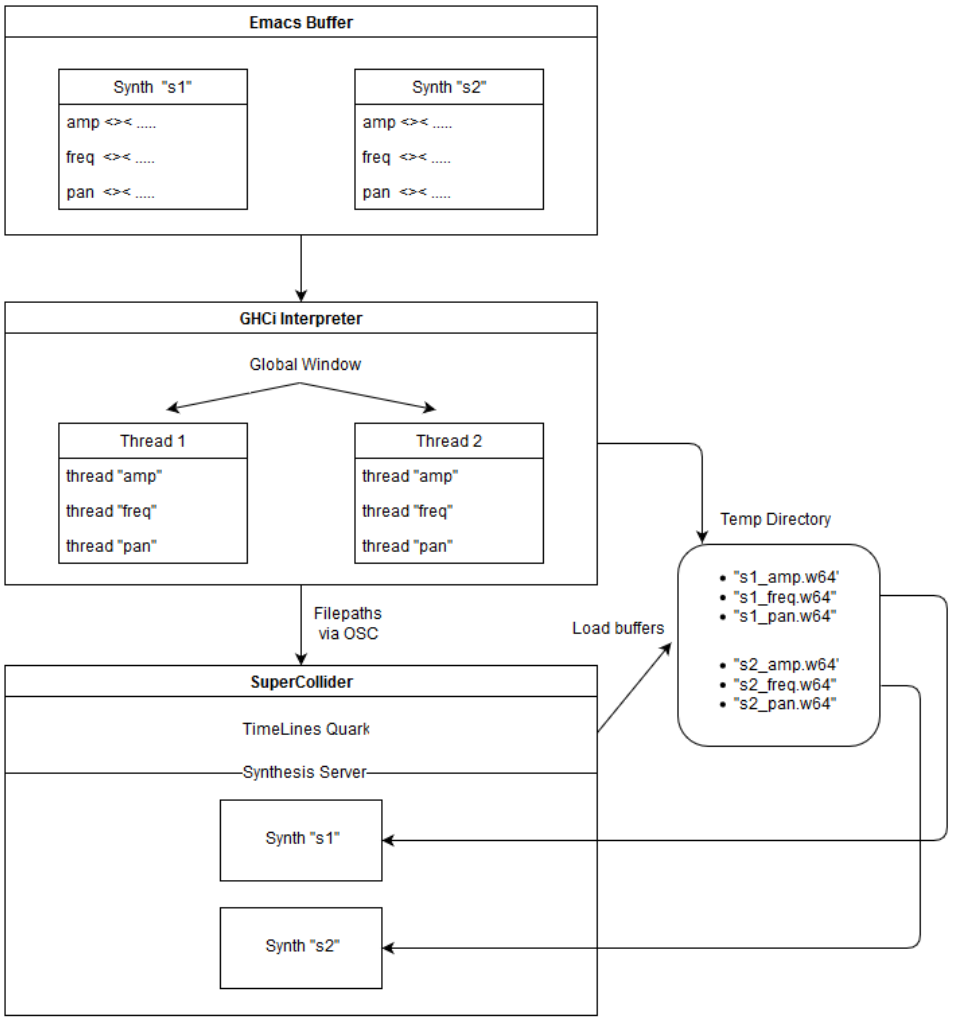


Figure 4.1: Diagram outlining the structure of TimeLines.

4.2.1 Error Handling

With every activity that involves writing code, especially under the pressure of real-time constraints and while being watched by an audience, errors should be expected and protected against appropriately. TimeLines handles errors by design: as there is no continuous process, other than the indexing of buffers and sound synthesis, running in real-time, any errors on Haskell's side will simply fail to produce the new buffers that will replace the old. As soon as a buffer has been loaded on the server it is stored in memory, making sure it will be available for indexing until a new buffer has successfully been written and is ready to replace the old. Therefore, any newly introduced errors in the Haskell code will be caught by Haskell's type checking system when the user attempts to evaluate the code and an error message will be printed on the console, all while the music keeps looping in the background without a glitch. Even a single error in a single thread's function will prevent the whole block from executing, thus making sure that all code will be executed only if the interpreter is sure that every buffer will be written successfully, so as to avoid changing all but a few of the server's buffers and causing unexpected and undesired combinations of parameter buffers.

On the side of SuperCollider, the only possibility for glitches is the case of a synth being instantiated with some of its buffers empty, which may be caused by those parameter equations being more computationally demanding to render and taking a few extra seconds. This has been solved by using a fade-in envelope for every synth's instantiation, making sure it does not produce any output until all buffers have finished rendering and have been loaded. Furthermore, in SuperCollider version 3.9 the `Sanitize` UGen was added, which monitors a signal for unwanted values like infinity, NaN, etc., and replaces them with a number, 0 by default. Since this version was relatively new at the time of development, it was not used in SynthDefs to preserve compatibility with users running older versions, but it remains as a possibility to make sure no unwanted values are plugged into sensitive UGens.

Chapter 5

TimeLines: Implementation

5.1 SuperCollider

As discussed in the previous chapter, the role of SuperCollider in TimeLines is twofold: firstly, it provides the sound processes that index the buffer data in real time and apply them to the various synthesis parameters. This is achieved exclusively by means of SynthDefs, which act as blueprint definitions for the various kinds of synths, including the one that behaves as the global time transport.

Secondly, it provides a stateful interface between the pure Haskell code and the server. This interface is responsible for keeping track of the amount and state of synths running on the server, as well as handling the incoming buffers and routing them to the synth and parameter they are supposed to be controlling. This frees Haskell to focus exclusively on creating the buffers when the user asks for them and allows for a simple, effective, and pure implementation.

5.1.1 SynthDefs

The most important SynthDef in TimeLines is that of the phasor that acts as a general playback transport, essentially guiding and synchronizing the simultaneous indexing of all control buffers. This transport system should include the following features:

- Looping when it reaches the end of its cycle. This looping behaviour can be used both in an offline production context, as well as an online live performance setting. In the offline case, it simulates the behaviour of selecting and looping a certain time window in a DAW, to allow the producer to constantly iterate changes over a section of a piece. In the second case, it can be used to keep the music playing while the performer is modifying and adding to the code, similar to TidalCycles' implicit cyclical nature of time.
- The option to toggle looping off. While this would be useful mostly in an offline activity, the user should be able to switch the phasor's looping off if they only want to audition a window

once and then work in silence.

- Following the above, the ability to silence the output of synths at the end of a non-looping cycle, even if their amplitude at that point remains above zero. Otherwise, synths whose amplitude is not equal to zero at the end of their current amplitude buffers would be left "hanging", producing sound until either freed or a new amplitude buffer is provided.
- The option to force the phasor to reset to its starting point. The user should be able to repeat the playback of the start of a window without having to wait for the phasor to complete its cycle, thus achieving a quicker iteration workflow.

```
  SynthDef(\timer, {
    |dur = 1, loopPoint = 1, t_manualTrig = 0, startPoint = 0|
        var sig = Phasor.ar(
                t_manualTrig,
                1/(dur*SampleRate.ir()),
                startPoint,
                loopPoint,
                startPoint
        );

        Out.kr(~silencerBus, sig);
        Out.ar(~t, sig.clip(0, 1));
}).add;
```

Code Snippet 5.1: Transport phasor SynthDef

The Phasor UGen proved to be the best way to implement such behaviour. Its arguments in order are: reset trigger, rate of increase per sample, start point, loop point, and reset point. Once instantiated, the phasor will start rising from the start point by the rate specified until it reaches up to but not including the loop point, at which point it will reset back to the start. If a trigger is received at any point in its cycle, the phasor will jump to the reset position and continue from there. In this case, the reset position is equal to the start, so sending a trigger to the phasor will force it to restart its cycle prematurely, similar to how double pressing the spacebar in most DAWs will stop the playback and start from the beginning of the selection.

The toggling of the looping behaviour is achieved by changing the looping point argument. By setting it equal to infinity, the phasor will keep rising indefinitely without looping. Before being output to the time bus, the phasor's output is clipped between 0 and 1, effectively stopping at a constant value of 1 when looping is turned off and the cycle has finished. At the same time, the unclipped version is output to a separate bus, where a value higher than 1 triggers the muting of all synths, in order to avoid "hanging" sounds when a synth's amplitude buffer ends with a non-zero value.

The silencing is achieved by means of another synth, which reads the phasor's unclipped output and silences every synths' output if the phasor crosses the threshold of a cycle's end. The resulting jump from 0 to 1 and vice versa is passed through a `Lag2` UGen which smooths the transition and produces a fade in/out effect.

```
SynthDef(\silencer, {
        var phasor = In.kr(~silencerBus);
        var reverbIn = In.ar(~reverbOut, 2);
        var dryIn = In.ar(~dryOut, 2);

        var switch = Lag2.kr(phasor < 1);
        reverbIn = reverbIn * switch;
        dryIn = dryIn * switch;
        Out.ar(~reverbSilencedBus, reverbIn);
        Out.ar(0, dryIn);
}).add;
```

Code Snippet 5.2: The silencer SynthDef, which mutes all synths when the transport is stopped at the end of a cycle.

In order for a synth to read and use the timer's phasor to index into its buffers, it simply has to input the normalised ramp and scale it by the number of samples each of those buffers has to be indexed.

```
SynthDef(\fm, {
        |amp, freq, ratio, index, pan|

        var t = In.ar(~t);
        var freq_  = BufRd.kr(1, freq, t * BufFrames.kr(freq));
        var ratio_ = BufRd.kr(1, ratio, t * BufFrames.kr(ratio));
        var index_ = BufRd.kr(1, index, t * BufFrames.kr(index));
        var amp_   = BufRd.kr(1, amp, t * BufFrames.kr(amp));
        var pan_   = BufRd.kr(1, pan, t * BufFrames.kr(pan));

        var fade = Env([0, 1], [~fadeTime], \sine).kr(0);

        var modFreq = freq_ * ratio_;
        var mod = SinOsc.ar(modFreq) * index_ * modFreq;
        var sig = SinOsc.ar(freq_ + mod) * amp_ * fade * fade;

        Out.ar(~reverbBus, Pan2.ar(sig, pan_));
}).add;
```

Code Snippet 5.3: An example SynthDef of a simple 2-operator FM synth

Here, keywords declared as arguments such as *amp*, *freq* etc. do not carry the actual value that should be assigned to those parameters, but rather the number of the buffer that stores those values. In order to extract the values and plug them where appropriate, a `BufRd` UGen is used to index into the provided buffers by reading the global time phasor, which has been normalised to have a range of $[0, 1)$, and scaling it by the number of frames each of those buffers has, provided by the `BufFrames` UGen. This allows for each buffer to have a different sampling rate, depending on the amount of temporal precision required for each parameter, while still being traversed over the same amount of time by the global transport phasor.

By convention, the resulting stream of values from reading each buffer is saved into a variable with the same name as the buffer appended with an underscore. Of course, users are free to adopt other naming conventions they are more comfortable with.

To demonstrate a slightly larger, polyphonic SynthDef, the following groups together four sine waves whose frequency and amplitude can be controlled independently, in addition to a group amplitude:

```
~noiseFreq = 0.8;
~lagTime = 0.9;
SynthDef(\sine4, {
        |amp, amp1, freq1, amp2, freq2, amp3, freq3, amp4, freq4|

        var t  = In.ar(~t);
        var amp_   = BufRd.kr(1, amp, t * BufFrames.kr(amp));
        var amp1_  = BufRd.kr(1, amp1, t * BufFrames.kr(amp1));
        var freq1_ = BufRd.kr(1, freq1, t * BufFrames.kr(freq1)).
            lag(~lagTime);
        var amp2_  = BufRd.kr(1, amp2, t * BufFrames.kr(amp2));
        var freq2_ = BufRd.kr(1, freq2, t * BufFrames.kr(freq2)).
            lag(~lagTime);
        var amp3_  = BufRd.kr(1, amp3, t * BufFrames.kr(amp3));
        var freq3_ = BufRd.kr(1, freq3, t * BufFrames.kr(freq3)).
            lag(~lagTime);
        var amp4_  = BufRd.kr(1, amp4, t * BufFrames.kr(amp4));
        var freq4_ = BufRd.kr(1, freq4, t * BufFrames.kr(freq4)).
            lag(~lagTime);

        var fade = Env([0, 1], [~fadeTime]).kr(0);

        var sin1 = Pan2.ar(SinOsc.ar(freq1_) * amp1_, LFNoise2.kr(
            ~noiseFreq));
        var sin2 = Pan2.ar(SinOsc.ar(freq2_) * amp2_, LFNoise2.kr(
            ~noiseFreq));
        var sin3 = Pan2.ar(SinOsc.ar(freq3_) * amp3_, LFNoise2.kr(
            ~noiseFreq));
        var sin4 = Pan2.ar(SinOsc.ar(freq4_) * amp4_, LFNoise2.kr(
            ~noiseFreq));

        var sig = (sin1 + sin2 + sin3 + sin4)/4 * amp_ * fade *
            fade;
        Out.ar(0, sig);
}).add;
```

Code Snippet 5.4: A 4-voice sine wave SynthDef with individual frequencies, amplitudes, and randomized pan.

The frequency of each sine wave is then "lagged", smoothly interpolating from one value to the next over a set amount of time to produce a gliding effect. This could of course be achieved by appropriately modulating the frequency values in the buffers, but since this SynthDef was specifically written for 4-voice chords that glide from one to the next, it made more sense to hide this functionality behind the scene as it would add one more level of complexity for the programmer to implement that.

Another behavioural characteristic of this SynthDef hidden behind its interface is the randomisation of the panning of each voice. Using the `LFNoise2` UGen, which produces quadratically interpolated random values at the interval specified, results in an ever-moving cohort of sine waves that travels the stereo field. Again, this could easily be achieved by providing access to a `pan` argument for each voice in the SynthDef, but that would attach an additional four lines of text to the end-user's interface and a further set of parameters to manage. Since this synth was specifically designed for the purpose of creating an evolving chordal soundscape which glides from one chord to the next, it made more sense to "bake" these features directly into its SynthDef and provide a more limited, *curated* even, set of parameters for the user to worry about and control in real time. We are thus allowing the instrument to have a larger influence on the sort of usage it gets from the musician, withholding some of the control so that the user can instead focus on what can be done with the instruments available, rather than what other instruments can be produced from them.

5.1.2 TimeLines Quark

In addition to providing the processes that synthesise all the sound, SuperCollider is also responsible for keeping track of all the running entities on the server and making sure that any incoming buffers can find their way to the appropriate parameter of the correct synth. This functionality is encapsulated in the TimeLines Quark, which is simply a class (in the OOP sense of the word) that can be installed on other systems either by downloading the code locally, or by automatically fetching it from SuperCollider's Quark git repository.

Once installed, the TimeLines class provides a convenient way of setting up the environment, both server- and language-side, to be ready for live coding. This setup, invoked by running `TimeLines.start` in the interpreter, includes in order:

1. Booting the server, if it is not already running.
2. Initializing the buses for the time phasor and reverb, as well as the language-side dictionaries and server-side groups. Two dictionaries are created, one for loaded buffers and one for

keeping track of running synths, and four groups in the order of: timer, synths, effects, and reverb.

3. Loading the SynthDefs and OSCdefs, which handle incoming OSC messages and call the appropriate functions. This will load and execute all valid `.scd` files present in the "defs" sub-directory of the Quark, which is where users can add their own.
4. Finally, instantiating the timer, silencer, and reverb synths, which are placed in the first and last node groups respectively. This is in order to make sure that any additional synths spawned by the user will be correctly placed in the server's node execution order and calculate their outputs after the time synth, but before the silencer or reverb synth.

The instance of the TimeLines class created during the process is stored in the `~timelines` global variable, so that it can be accessed by the user and other functions at any point.

After the server and language environments have been set up, the user can create a synth by using the Haskell interface, which writes the buffer files and sends their absolute filepath via an OSC message to SuperCollider. The running TimeLines instance then parses that filepath and extracts some useful information, such as the name of the synth the buffer belongs to, the SynthDef of that synth, and the argument of that synth the buffer is supposed to be controlling. By convention, the full filename of that buffer is these three pieces of information separated by an underscore. So, as an example, if a buffer belongs to the synth named "bass", which is an instance of the "lpfSaw" SynthDef, and contains the data that should be used to modulate the "freq" parameter, then that buffer's full name will be "bass_lpfSaw_freq", which can be converted to the actual filename by adding the ".w64" extension.

```
1  loadBuffer { |path|
2                var filePath = path.asString;
3                var pathList = path.asString.split();
4                var buffName = pathList[pathList.size - 1].split($.)
                    [0];
5                var info = buffName.split($_);
6
7                var synthName = info[0].asSymbol;
8                var synthDef = info[1];
9                var synthParam = info[2].asSymbol;
10               var synth = synthDict[synthName];
11
12               if(synth.isNil, {synthDict.add(synthName -> Synth(
                    synthDef, target: synthGroup))}, {});
13
14               bufferDict[buffName].free;
15               bufferDict.add(buffName -> Buffer.read(server,
16               filePath,
17               action: {|b|
18                 synthDict[synthName].set(synthParam, b)}));
19       }
```

Code Snippet 5.5: The function that gets called every time a new buffer is written.

Once the function extracts those three key pieces of information from the filepath, line 12 checks to see whether a synth with that name already exists in the dictionary of known running synths. If not, it creates one, using the synthdef specified, and places it in the synth node group. Line 14 then sends the "free" message to the corresponding entry in the buffer dictionary, which will free the buffer from memory if it already exists and will ignore the message otherwise. Finally, the new buffer is loaded on the server, and a callback is provided to update the synth once the loading has finished.

This function is core for the operation of TimeLines, as it acts like a bridge between the stateless world of pure Haskell functions and the necessarily stateful reality of controlling the server. Since Haskell refuses to keep track of what buffers and synths exist and may need freeing at any time, SCLang obliges to take care of this "impure" task and let Haskell concern itself with the creation of information, rather than its book-keeping.

5.2 Haskell

The role of Haskell is to provide the interface functions that are used to construct and combine signals, as well as the background functions that evaluate and sample those signals and write them to a file, ready to be loaded by the server.

5.2.1 Types

Whereas Object-Oriented programs are built around the construction of classes and their methods, the core of functional programs revolves around their data types and the functions that operate with them. The definitions of those types not only model and represent the problem domain of the program, but also set the guidelines for how other functions are to use them.

Following one of the goals initially set for this project, simplicity in both use and implementation were key.

Initially, the base types describing the raw values TimeLines would be working with, namely the `Time` type of the input to signal functions and the `Value` type of their output, were defined as type synonyms to Haskell's standard data types:

```haskell
type Value = Double
type Time = Double
type Window = (Time, Time)

type SamplingRate = Int
type SynthID = String
type Param = String
```

Code Snippet 5.1: Base data types, defined as synonyms to Haskell's types.

The `type` declaration in Haskell creates a synonym of a data type, making it clearer to read and reason about the type signature of functions, while at the same time guaranteeing that the synonym type will always be interchangeable with the original.

Initially, `Time` was defined as a `Rational` number. In Haskell, Rationals are defined as `Ratio Integer`, a ratio of two `Integers` whose size are only subject to available memory. Therefore, a `Rational` representation of `Time` would allow for extreme precision in its usage and calculation. However, the specification of the `Ratio` type requires the secondary type to be a member of a given typeclass before the compound type can be too. As the `Integer` type is not a member of typeclasses like `Fractional` or `RealFrac`, then a `Ratio Integer` cannot be used with functions that operate on such values, such as `floor`, `ceil`, or `sin`. Therefore, in order to be able to utilise the breadth of numerical functions available in Haskell, a representation of `Double` was chosen instead for both `Time` and `Value`, which has the added benefit that requires no conversion before being written to the final buffer file.

The `Window` type uses a `Tuple` of two `Time` values, marking the start and end points, to represent a temporal segment. During a TimeLines session, the current `Window` is always stored as a global reference so that, when the user asks for the sampling of a signal into a buffer, the sampler function knows which moments in time to evaluate the signal for. Finally, `SamplingRate` is an integer representing the rate at which a signal is to be sampled, while `SynthID` and `Param` types are simply `Strings` that denote the name of a synth and its desired parameter to control respectively.

Next comes the `Signal` type, the fundamental type representing all musical content in TimeLines. Signals were defined using Haskell's `newtype` declaration, which creates a type synonym that is not interchangeable with the original type and is considered unique, while still guaranteeing the same treatment by the compiler in terms of efficiency and optimisations.

```
newtype Signal = Signal {runSig :: Time -> Value}
```

Code Snippet 5.2: The Signal data type.

A value of type `Signal` is then created by providing a function from `Time` to `Value`. Haskell's record syntax here conveniently attaches a name to that, so that `runSig someSignal` automatically retrieves `someSignal`'s signal function, from which a final `Value` can be calculated by evaluating it for a `Time` argument.

There are four main types of signals to be distinguished:

- *Constant*: These signals always return the same value, no matter what point in time they are evaluated for.
- *Identity*: An identity signal will always return the value of time for which it was evaluated.
- *Periodic*: A periodic signal will produce a pattern that repeats over a certain amount of time, the most simple of which being a phasor or a sinusoidal wave.
- *Arbitrary*: An arbitrary signal can be created to return any stream of values, which may or may not repeat for any number of times and may be made to simulate randomness or chaotic systems.

Out of these, the identity signal is arguably the most important for the interface of TimeLines, as it is the only way for the flow of time itself to be imported into the signal domain and represented as a value that functions which operate on signals can work with. Without it, every signal would have to be explicitly created using a *lambda*, or anonymous, function in order to get some sort of reference to time, as the design of TimeLines and Haskell's lazy evaluation do not allow for a variable that *actually* changes over time, only conceptually. Due to its importance and commonplace usage, the identity signal was assigned to a function denoted by the letter 't', so that it can be used anywhere in the parameter equations to act as a placeholder for the actual passage of time:

In addition to the identity signal, constant signals are also very commonplace in TimeLines,

```haskell
t :: Signal Value
t = Signal $ \t -> t
```

Code Snippet 5.3: The identity, or time, Signal 't'.

since not every value need be modulated over time. Therefore, a function for producing constant signals was created by ignoring time itself alltogether and always returning the value itself:

```haskell
constSig :: a -> Signal a
constSig v = Signal $ \t -> v
```

Code Snippet 5.4: A function producing a Signal that always returns the same value.

Finally, the above types were put together to define the `TimeLine`, a type representing all the necessary information that the sampler function needs in order to produce the final buffer file that will be sent to the synth:

```haskell
data TLinfo = TLinfo {infWindow :: Window, infSR :: SamplingRate, infParam :: Param}
  deriving (Eq, Show)

data TimeLine = TimeLine {tlSig :: Signal, tlInfo :: TLinfo}
```

Code Snippet 5.5: Compound data types used to bundle together all the necessary information for sampling.

These types were defined using Haskell's `data` keyword, which allows the user to create any arbitrary algebraic data type as a combination of others. Record syntax was used here too, in order to avoid having to define getter functions for each and every component of these types.

5.2.2 Type Classes

Haskell's type system gains a lot of its power and flexibility from type classes, not to be confused with the notion of a class in OOP. A type class defines an interface for a certain kind of behaviour of types, which may have different implementations across different types but ultimately produces similar results. The simplest example is the `Eq` type class, which includes all types whose members allow for a way to test equality between them. For some types, such as `Integer` or `String`, testing for equality is straightforward, while for more complex types such as `Person`, which may include a name, height, and date of birth, testing for equality involves testing each field respectively and combining the results. Another example of a type class, one which enables much of the user interface functionality of TimeLines, is `Num`. Type members of the class `Num` can be thought of as numbers and offer instructions on how to perform numerical operations on them, such as adding, multiplying, negating, and taking their absolute value. The type classes whose implementation will be discussed in this chapter are the family of `Functor`, `Applicative`, `Monad`, and finally `Num`.

An extremely powerful and useful type class in Haskell is that of a `Functor`, which draws its inspiration from category theory of mathematics. By making a type an instance of a `Functor`, one specifies how that type is to be mapped over by other, normal functions. In other words, if one has a container type `MyValue a` (where a is a type parameter) and a function of type `a -> b`, then that type's `Functor` implementation will tell the compiler how to apply that function on the container's contents in order to return a value of type `MyValue b`, leaving the container's type intact while operating on its contents. The simplest and most commonly used example of a functor is that of a list, which specifies that mapping a function over its contents is equal to a list, whose elements are the result of applying that function to each of the original list's elements. The minimal complete `Functor` definition for a type `f` demands the implementation of the function `fmap::(a -> b) -> f a -> f b`.

```
instance Functor Signal where
  fmap f (Signal sf) = Signal $ \t -> f (sf t)
```

Code Snippet 5.6: Signal's definition as an instance of Functor.

By carefully deconstructing the type signature of `fmap`, and keeping in mind the definition of a `Signal`, we can read the above as: "The result of applying a function `f`, which takes a value of type `a` and returns a value of type `b`, to a signal with a signal function `sf`, which takes a value of `Time` and returns a value of `a`, is a signal with a signal function which takes a value of `Time` and returns a value of type `b`, namely the result of applying `f` to the value `a` returned by the original signal function, `sf`." In simpler terms, mapping a normal function over a signal at any point in time is achieved by evaluating the signal for that moment, passing the resulting value (which is no longer a `Signal` but rather a base type) to that function, and wrapping it back into the `Signal` container.

The usefulness of this may seem unclear for now, but it will become apparent when we try to hide the complexity of signals being value containers behind the scenes, allowing the user to treat them as normal numerical values without having to worry about their implementation.

Next in the hierarchy is the `Applicative` class, which demands at least two function implementations for a complete definition: `pure` and either `liftA2` or `<*>`.

```
instance Applicative Signal where
  pure = constSig
  liftA2 f s1 s2 = Signal $ \t -> f (runSig s1 t) (runSig s2 t)
  sf <*> s = Signal $ \t -> (runSig sf t) (runSig s t)
```

Code Snippet 5.7: Signal's definition as an instance of Applicative Functor.

The function `pure::(Applicative f) => a -> f a`, simply "lifts" a normal type to be any Applicative container of that type. Since in the case of `Signal` that has already been described and implemented as a constant signal, which takes any value and returns a signal that always evaluates to that value, then `pure` can be defined to be equal to `constSig`.

Another useful function is `liftA2::(Applicative f) => (a -> b -> c) -> f a -> f b -> f c`, which instead of simple values it "lifts" two-argument functions, allowing them to work with the contents of any two Applicative containers and return a third. By adding `liftA2` to a type's arsenal, that type can now utilize any normal binary function to operate on its own contents (provided that the types of those contents match the types expected by the function) and automatically wrap the result in the same container type. This is achieved for `Signal` by first evaluating the two signals for the current moment in time, in order to extract the base type values that the function expects, applying the function to those two resulting values, and wrapping the result of that back into a `Signal`. The implementation of `<*>` is quite similar, although its semantics differ in that it applies a *time-varying function* to a `Signal`, which in itself is a *time-varying value*. Therefore, to apply the resulting function to the resulting contents of the signal, one has to first evaluate the function-signal and then apply its output to the result of the value-signal.

Last in this family of type classes comes the `Monad`, which allows a function that needs a base type to extract it from a container and use it to produce another container, essentially allowing for temporarily extracting a value from its context with the condition that the result will again be inside the context. For example, say we have a `Signal Double` and an arbitrary function that takes a `Double` and returns another `Signal Double`, such as the `constSig` function or similar. The monad's (≫=) operation, read as `bind`, allows for threading contextual operations together, essentially saying "use this monad's content as a pure value to create another monad".

The result of this operation is a `Signal`, which first evaluates the original signal, passes the result to the second signal-generating operation, and evaluates that too, finally wrapping it back inside the `Signal` context.

```
instance Monad Signal where
  return = pure
  firstSig >>= f = Signal $ \t ->
    let resultingSig = f $ runSig firstSig t
    in  runSig resultingSig t
```

Code Snippet 5.8: Monad instance implementation for Signal.

Finally, perhaps the most important instance of the `Signal` data type is that of the `Num` type class. To appreciate its contribution to the workflow and user experience of TimeLines, one needs to take a closer look at Haskell's type inference system. The concept of polymorphism allows for functions to operate with a whole range of types, as long as those types have provided valid instance definitions for all classes required by a function. For example, the addition operator `(+)` has a type signature of `Num a => a -> a -> a`, specifying that as long as a type is a member of the `Num` class, then it can add two of them together and return a third. Similar class requirements are used by all other numeric operators, as well as the majority of Haskell's numerical functions.

Due to Haskell's lazy evaluation, the type inference system will wait until absolutely necessary to infer the type of a value, propagating all its possible types until one of them is deemed necessary by the context. As an example, in the case of adding two values by typing `2 + 2`, the compiler will only infer that the type of the values, and thus the type of the resulting value too, is an instance of `Num`, but will refrain from assuming whether it is `Int`, `Double`, `Rational` etc. If we were to change one of the two arguments to a fractional value, say `2.5`, then the compiler would specify even further by assuming both types are numbers *and* members of the `Fractional` type class, but would still not impose neither `Double` nor `Float`. If the result of this addition was to be provided to a function that expects a `Double`, then the compiler would finally assume their actual type is `Double` and proceed accordingly.

If signals were not a member of the `Num` class, they would not be able to be passed as arguments to anything that expects to work with numbers, requiring that even the simplest of numeric functions and operators be replaced or completely rewritten to accept and return signals. To make matters worse, numerical literals would have to explicitly be transformed to signals by use of `constSig`, adding an additional burden on the programmer and visually polluting the code with a function call for every numeric literal not explicitly passed to a signal generator. Fortunately, all of this can be avoided by telling the compiler how to treat the `Signal` type as a `Num`, thus taking full advantage of the language's lazy type inference system to do all the work for us.

```haskell
instance (Num a, Eq a) => Num (Signal a) where
  negate      = fmap negate
  (+)         = liftA2 (+)
  (*)         = liftA2 lazyMul -- lazy multiplicator
  fromInteger = pure . fromInteger
  abs         = fmap abs
  signum      = fmap signum
```

Code Snippet 5.9: Signal's definition as an instance of Num.

The type class constraints dictate that a `Signal` is a `Num` only if the type that the signal carries is a `Num` itself, in addition to being testable for equality. Notice the use of `fmap` and `liftA2`, which respectively allow unary functions, like `negate` and `abs,` to operate on the contents of a `Signal,` and binary functions, like `(+)` and `(*)`, to be lifted to work with two `Signals` and return a third.

You may also notice that multiplication is defined as the lifting of `lazyMul`, instead of the `(*)` operator one might expect. This is a custom function that tries to take full advantage of Haskell's laziness by providing a multiplication operator that will only evaluate its second argument if the first argument is non-zero:

The reason for this is simple: as discussed in Chapter 3, arbitrary sequences of musical content can be formed by summing multiple different functions, each of which is "activated" for only those

```
lazyMul :: (Num a, Eq a) => a -> a -> a
lazyMul 0 _ = 0
lazyMul a b = a * b
```

Code Snippet 5.10: Signal's definition as an instance of Num.

moments in time we have specified and ignored for all others, thus creating the illusion of a unified, sequencial motion. This deactivation happens by means of multiplication with zero, essentially nullifying the contribution of that function towards the sum for all points in time before and after its allocated slot. The value which evaluates to zero for those moments, usually placed at the LHS of the multiplication, is a simple bounds test and therefore easy to compute, however the function that gets nullified may be of an arbitrary level of complexity, as it may itself consist of a large sum of functions arranged throughout its duration. Therefore, by first checking the first argument of the multiplication, the switch-function, to see whether it evaluates to zero, we can save the processing resources that would be used to calculate the content-function, which would be completely wasted as its result would be flattened to a constant zero.

5.2.3 Functions

The three main utility functions that a user interacts with during a typical TimeLines session are `window`, `synth`, and `sendParam`.

The `window` function takes two arguments, both of a value `Signal Time` and assumed to be constant for all times, and uses them to change the current window and update the transport synth on the SuperCollider server. The window is stored as an `IORef`, a parameterized built-in Haskell type which creates a mutable reference to a value of any type, in this case `Window`. Once the `window` function receives its arguments, it updates the `IORef` with the new start and end values and proceeds to send an OSC message to the server with the duration of the new window, since that is the only thing required to make sure the timer phasor indexes through the buffers at the appropriate rate.

The `sendParam` function receives the signal that will control a parameter, and a `Param` (i.e. `String`) which specifies the synth's argument that the signal should be assigned to. It then returns a `ReaderT SynthID IO ThreadId`, which is Haskell's way of describing a type that, once provided with a value of `SynthID`, will return an action of type `IO ThreadId`. In other words, before `sendParam` can proceed with sampling the provided signal and writing it to disk, it needs to be provided with a context of type `SynthID` (i.e. `String`), in order to know the name and SynthDef of the synth that the resulting file belongs to and prepend them to its filename. Once this `SynthID` has been provided, then the buffer file can be written and its absolute filepath sent to the server for loading.

```
sendParam :: Param -> Signal Value -> ReaderT SynthID IO ThreadId
sendParam p sig = do
  synthName <- ask
  let filepath = synthName ++ "_" ++ p ++ ".w64"
  liftIO $ forkIO $ do
    writeParamFile filepath sig
    sendUpdateMsg filepath

(<><) = sendParam
```

Code Snippet 5.11: The function responsible for assigning a signal to a parameter.

Notice the monadic function `ask`, which extracts the `SynthID` from the provided context and binds it to a variable name. Furthermore, the call to `forkIO` spawns a new processor thread which executes the following actions, namely writing the file and sending a message to the server. Lastly, the `sendParam` function is bound to the custom `(<><)` operator for aesthetic reasons, as it allows to have all parameter names lined up to the left and all the signals to the right of the operator.

The `writeParamFile` function handles the actual rendering of a given signal and its writing to the provided filepath. It proceeds to read the current Window and generate a `TLinfo`, which contains information such as the start and end points of the sampling and the total number of values to be produced, which depends on the sampling rate. A default sampling rate of $700Hz$ was chosen, as it matches SuperCollider's default rate for control-rate UGens and is enough to represent modulation of up to $350Hz$, which is well inside the audio rate. Nevertheless, the default rate can be changed and different rates can be used for different parameter buffers, with the server making sure they are indexed over the same amonut of time.

The resulting file, which is of a 64-bit double wav format with the extension ".w64", is written using a Haskell interface to the C library libsndfile, which is the same library used by SuperCollider for the reading and writing of its own audio files. This ensures no compatibility issues with what is written by TimeLines and what is read by SuperCollider.

The more important of the utility functions is the one that samples the provided signal over the given window. The list of resulting values is by mapping the evaluation of the signal over a list of the domain's `Time` values. This domain consists of the discrete range of `Time` values, in seconds, between the window's start and end points, incrementing by the inverse of the sampling rate.

```
getVals :: TimeLine a -> [a]
getVals (TimeLine sig info@(TLinfo (s, e) _ _)) = map f domain
  where f = runSig sig
        domain = fromToIn s e $ infNumFrames info
```

Code Snippet 5.12: Obtaining the sampled values of a signal by mapping it over the values of its observed time domain.

This mapping operation provides a simple and elegant way to sample the contents of a signal over the desired window of time, purely functionally and in parallel.

Lastly, the outermost interface function is `synth`, which provides the `SynthID` context to all subsequent calls to `sendParam`. Each call to `synth` begins by spawning a new thread, so that multiple synths from a block can be processed in parallel. It then proceeds to get the path to the system's temporary directory, where a `TimeLines/buffers/` directory has already been created, append the `SynthID` and provide it as the context for all following parameters to use as the base filepath for their buffers.

```haskell
synth :: SynthID -> ReaderT SynthID IO ThreadId -> IO ThreadId
synth synthID params = forkIO $ do
  pathToTemp <- getTempDirectory
  runReaderT params $ pathToTemp ++ synthID
  return ()
```

Code Snippet 5.13: The Synth function which provides a context for parameters.

Functions for use during a session were implemented similarly to their description in Chapter 3, just within the Signal datatype. They can be found in the `Instruments.hs` file.

5.3 Emacs

Customising the Emacs editor is achieved via configuration files written in Emacs-Lisp. For the purposes of TimeLines, two such files have been written: one that configures the general look and behaviour of the editor, and one that provides the major mode for working with TimeLines sessions.

5.3.1 General Configuration

As soon as Emacs boots, it loads and executes the initialisation file. This file is responsible for executing all commands that customise the editor, including changing settings, installing and configuring packages, and more. By utilising the `use-package` extension, one can simply specify the packages that should be used, along with any configuration commands that should be run for those packages. If they are not already installed on the system, they will automatically be downloaded and installed before the initialisation completes. This means that a brand-new installation of Emacs can be configured to look and function exactly as a user wants, simply by providing the appropriate initialisation file, and given that an internet connection is available for downloading all missing packages. Therefore, anyone can obtain the initialisation file for TimeLines' Emacs configuration and have the editor automatically assemble itself to match it.

Some of the commands executed at initialisation include stripping Emacs of unnecessary and distracting elements, such as the menu and tool bars and the welcoming screen, changing the theme to a more comfortable and visually appealing dark theme, enabling the highlighting of the cursor's current line, etc. All these customizations aim to rid the editor of visual clutter and add functionality which will help the user find their way around the code as quickly and efficiently as possible.

5.3.2 Navigation

During a live coding session, a large part of one's interaction with the keyboard involves navigating around the code. In most conventional editors, this is achieved by using the keyboard's arrow keys on the right-hand-side of the letters, which forces the user to move their hand away from the part of the keyboard where the actual typing happens. This does not only waste time moving between separate parts of the keyboard, it oftentimes also consumes valuable moments of concentration to visually locate the arrow keys and letter keys respectively. TimeLines reduces these distractions by allowing one to navigate around the text without moving the hand away its natural resting position on the keyboard.

This is achieved by changing the keyboard shortcuts of Emacs' global keymap, which available from anywhere in the editor. The following lines remove any functionality assigned to these combinations of Control, Meta (i.e. "alt"), and letter keys and instead assign functions that move the cursor:

```
(global-unset-key (kbd "C-j"))
(global-set-key (kbd "C-j") 'left-char)

(global-unset-key (kbd "C-l"))
(global-set-key (kbd "C-l") 'right-char)

(global-unset-key (kbd "M-j"))
(global-set-key (kbd "M-j") 'backward-word)

(global-unset-key (kbd "M-l"))
(global-set-key (kbd "M-l") 'forward-word)
```

While idly resting on the keyboard, a user's right-hand index and ring firgers will naturally sit on top of the 'j' and 'l' keys, making them the ideal candidates for moving left and right in text respectively. By following the common Emacs convention, the Meta version of the combination will have similar but extended functionality to the Control version. In this fashion, "C-j" moves a character to the left while "M-j" moves a whole word to the left. Similarly, the rest of the shortcuts were remapped to essentially move the arrow keys under the resting fingers.

To avoid constantly having to twist one's wrist in order to reach the control key, which can lead to repetitive strain injuries commonly referred to by users as "Emacs pinky", the caps lock key was modified to act as another control key, therefore being immediately reachable to the left of the 'a' key, the resting position for the left hand's pinky finger. This customization can be achieved in all three major operating systems, with Apple's OS X and many Linux distribution providing an option for this in the system settings, and via a one-line command for AutoHotkey in Windows, a free keyboard remapping utility.

To speed up navigation around the text even more, the default values for the keyboard's keypress repetition delay and interval were modified. The key repetition delay represents the time between a key is pressed down and when the system starts repeating the functionality of that key, whether that is typing a letter or moving to the next character. The interval of that repetition then defines how quickly that functionality will be repeated, with lower rates resulting in more triggers of that functionality over time. The initial delay was set to 230 milliseconds and the repeat rate to 30 repetitions per second, resulting in a much responsive behaviour from the keyboard which was still controlable after getting used to.

Lastly, even though the Dvorak keyboard was chosen as a less strenuous typing layout, care was taken to make sure all keyboard shortcuts would be layout-independent and thus identical for a user favoring the more commonplace QWERTY layout. This was achieved by adding a key translation layer to Emacs' global key map, which remaps single letter keys to their Dvorak counterparts but leaves any shortcuts involving those keys unaffected. For example, the following lines remap the 'j' and 'l' keys to type 'n' and 's' instead, without however affecting any shortcuts using those keys that have been defined over the QWERTY layout.

```
(define-key key-translation-map (kbd "j") (kbd "h"))
(define-key key-translation-map (kbd "l") (kbd "n"))
```

The rest of the keyboard was remapped in the same fashion.

5.3.3 TimeLines Mode

The functionality specific to the operation of a TimeLines session was grouped into a major mode. This mode was derrived from Haskell's own mode, which provides features such as syntax highlighting and automatic indentation, and based on the Emacs mode for the TidalCycles language, since they are both Haskell-based and thus share much of the functionality.

This mode, which gets automatically triggered when the user opens a file with the "*.tl" extension, is responsible for starting and quitting the GHCi interpreter process, sending the code blocks as strings for evaluation, and printing the interpreter's output in a secondary buffer. It also implements all TimeLines-specific shortcuts, such as resetting the server, restarting playback, and toggling the transport's looping on and off.

Chapter 6

TimeLines: User's Guide

As with any custom software, the installation and configuration processes can prove to be tricky. For this project, great care was taken to ensure that everything can be set up to work together by simply installing all components and downloading their respecive configuration files.

This chapter aims to provide the new user with a brief guide for the installation, configuration, and usage of TimeLines.

6.1 Installation and Configuration

TimeLines depends on the installation of its three software components, namely SuperCollider, Haskell, and Emacs. To aid in the installation process and for future updates, Git should also be installed. Once all three components have successfully been installed, they can be configured by pointing them to the appropriate configuration files. These are:

- SuperCollider: the TimeLines quark, along with the SynthDef and OSCdef files. These can be found and cloned at `https://github.com/lnfiniteMonkeys/TimeLines-SC`. It is recommended that the repository gets cloned to `~/supercollider/timelines`, but any other path will work too. Once downloaded, they can be installed by evaluating 'Quarks.install("~/path/to/the/timelines-quark")' in the SuperCollider interpreter, replacing the string for the actual path to the cloned repository.
- Haskell: The TimeLines source code has an external dependency on the "libsndfile" C library. Multi-platform installation instructions can be found on its website, `http://www.mega-nerd.com/libsndfile/`. Once it has been installed, the source code for TimeLines can be found and cloned at `https://github.com/lnfiniteMonkeys/TimeLines`. It is recommended to place the repo in `~/timelines`, as that is where GHCi will look for it. Once cloned, it can be installed by navigating to its directory and running `cabal install` in a terminal.

- Emacs: Once Emacs has been installed, it can be automatically configured by placing the TimeLines configuration files in its startup directory, usually located at `~/.emacs.d` (it is recommended to delete the `~/init.el` file that automatically gets created in the home folder and instead use the startup directory). These files can be cloned from `https://github.com/lnfiniteMonkeys/TimeLines-Emacs`. Once placed inside Emacs startup folder, simply restart Emacs and all necessary packages will get automatically installed (provided there is access to the Internet).

6.2 Usage

After all necessary components have been installed and configured, it is time to boot everything up. Firstly, start the SuperCollider TimeLines quark by evaluating `TimeLines.start` in the IDE. This will also start the server if it is not already running. You can then minimize the IDE and open Emacs. Open a new buffer (C-x C-f) with the extension ".tl" and save it somewhere. A session can then be started by pressing "C-c C-s", which will open an output window and load the source code.

6.2.1 Shortcuts

The following diagram illustrates some of the key shortcuts for using TimeLines, irrespectively of whether the QWERTY or Dvorak layout is used. For shortcuts containing more than three keys, only the first key has to remain pressed throughout.

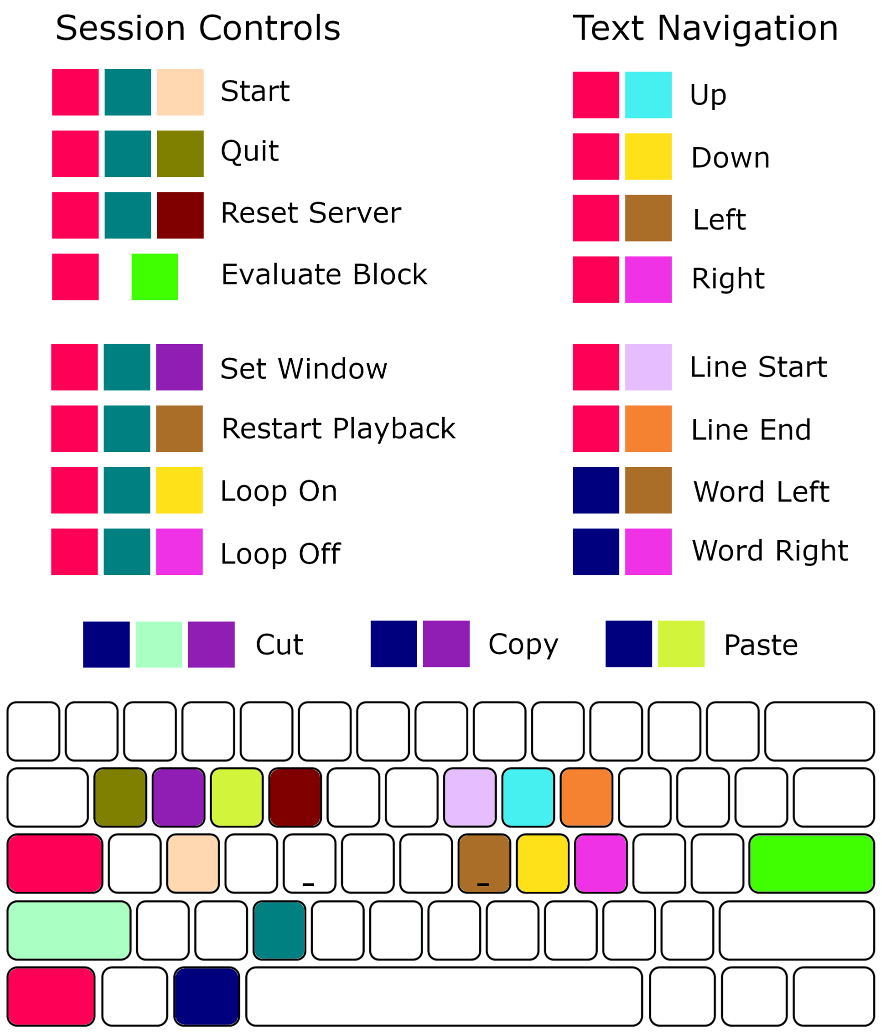


Figure 6.1: Colour-coded keyboard shortcuts for TimeLines.

6.2.2 Functions

Once an interactive session has started, now try making some sound. Type "do", followed by "enter", and then "fm", followed by the space bar. The resulting text should look something like the following:

```
do
  synth "_fm" $ do
    "amp" <>< let
      in 0
    "freq" <>< let
      in 0
    "ratio" <>< let
      in 0
    "index" <>< let
      in 0
    "pan" <>< let
      in 0
```

with the cursor located right before the underscore. Type a name, for example "s1", and then press "tab" to navigate to the other parameters and give them appropriate values. Once ready, press "C-<RET>" to evaluate the block. The synth's sound should fade in shortly after.

To change the window and get some higher-level structure, insert a new line at the top of the "do" block and type "phasors", followed by the space bar. This will paste a snippet that allows to set the bpm, number of beats, and number of bars of the window. It then provides a set of useful phasors that represent individual beats and whole bars. This should look something like:

```
do

  let (phBeat, phBar, beatDur, barDur, totalDur) = bpmToPhasors bpm numBeats numBars

  window 0 totalDur

  synth "s1_fm" $ do

    "amp" <>< let

      in 0.1

    "freq" <>< let

      in 300

    "ratio" <>< let

      in 1

    "index" <>< let

      in 10

    "pan" <>< 0
```

Use the "tab" key to navigate to the next arguments and replace them with the desired values. Make sure no blank lines are within the do block, as that will only execute part of the code.

Lastly, to introduce some rhythm and melody, we can use the `lerp` and `semi` functions for the amplitude and frequency respectively. The following modulates the amplitude by an exponential ramp that goes from 1 to 0 twice every beat, while a melody is plyaed by indexing through a list of semitones every two bars:

```
do

  let (phBeat, phBar, beatDur, barDur, totalDur) = bpmToPhasors 120 4 8

  rhythm = pow 2 $ lerp 1 0 $ fast 2 phBeat

  melody = semi $ fromList [0, 2, 5, 7, 8, 7, 10, 5] $ slow 2 $ phBar

  window 0 totalDur

  synth "_fm" $ do

    "amp" <>< let

      in 0 + 0.2*rhythm

    "freq" <>< let

      in 300 * melody

    "ratio" <>< let

      in 1

    "index" <>< let

      in 10

    "pan" <>< let

      in 0
```

More useful functions can be found in the "Instruments.hs" file, located in "~/timelines/-Sound/TimeLines/". They can be edited and new functions can be added there, just make sure to save the file and reload the source code by pressing "C-c C-l".

To add custom snippets, you can simply save them in "snippet/timelines-mode" in the Emacs home directory. For further information you can refer to the documentation for the `yasnippets` package.

Chapter 7

Testing and Evaluation

The creative and long-term nature of many of this project's goals make it inherently difficult to test towards. Firstly, the evaluation of its potential as a flow-inducing, scalable musical instrument requires prolonged practice and the development of a certain level of mastery of its abilities. Furthermore, its unique skillset and approach to thinking about and creating music makes it hard to compare against the vastly different skillset of most other live coding environments.

Nevertheless, the success of the project was tested in a series of live and taped performances, the variety of musical outcome that could be achieved during those performances, and the resulting levels of flow experienced.

7.1 Live Performances

The first performance was in the context of a 2-day "Music Hack Day" Hackathon held at Goldsmiths, University of London, at April 7th 2018. This was still the early prototype stage for TimeLines, while its structure was still different to its current state and the `Signal` datatype was a simple synonym for `Double`. This meant that all parameter equations had to be provided a lambda in order to get some reference to time, which in turn meant that time could only be local to that parameter and that it had to be explicitly passed around as an argument for functions. In addition, due to the early stage of development at the time, it was my first time playing with it for more than 10 minutes in a row, and completely improvised with instrument functions that were written an hour or two before the performance. Nevertheless, I managed to perform for 30 minutes while immersed in the activity simply by thinking about what it was I wanted to achieve with the music and working out a way to represent that mathematically using the handful of functions I had available.

The whole set was performed with a total of six sine waves, arranged into three two-operator FM synthesizers. Even with this limited setup, I never felt that the available sonic palette was limited, simply because the output of those synths could change drastically by applying different modulating functions to the modulation ratio and index parameters, as well as the usual amplitude and frequency. If anything, the limitation of only having these simple synthesizers to work with resulted in thinking of creative ways of assembling their parameters together over time. It also removed the temptation of constantly tampering with the sound process while a repetitive loop is playing in the background, therefore forcing me to engage more with the musical content and less with the technicalities of the synthesis processes.

While this performance followed a free-form approach by combining square and sine waves of various seconds-based frequencies and phases, in subsequent performances I wanted to experiment with more organized, grid-based structures. Using the newly developed `bpmToPhasors` function, which provides looping phasors for individual beats and whole bars, I attempted to achieve a more rhythmic, techno-inspired musical outcome. New SynthDefs were also written for this piece, using a sawtooth wave passed through an overdriven resonant low-pass filter.

Lastly, the longest performance of a duration of a full hour was based around 4-voice chord progressions and the usage of randomness to index into given scales. In contrast with other environments, TimeLines can easily use and reuse all sorts of constrained randomness by simply modulating the location at which the pseudo-random landscape is indexed. This allowed for creatively putting together musical material as a collage of random values that can be repeated, manipulated in intricate ways, and revisited later during the performance.

The feedback received from these performances, both from my personal experience of using the instrument and from later comments and suggestions of the audience, led to a restructuring of the initial design and to the current definition of the `Signal` datatype, which allows for global signals to be read by any synth, a major improvement in workflow and musical abilities.

7.2 Musical Flexibility

Despite the uniform and particular approach of TimeLines towards time-varying patterns, the range of musical workflows and outcomes that are supported is wide and diverse.

First and foremost, TimeLines is equally suited for both the stage and the studio. Due to its flexible approach regarding indexing different parts of a piece's temporal domain, a looping phasor can be used to achieve continuity during a live performance, while individual linear phasors can be used to audition certain areas of a piece while working in the studio. Due to the deterministic nature of pure functions, one can always be certain that the same code will realibly produce the same results, even when pseudo-randomness is heavily used.

Discrete and grid-based approaches to rhythm and melody are supported by means of quantizing values, whether of time or frequency. One is allowed to think in terms of bars, beats, and sections, with various phasors allowing to quantize the indexing of values to fit those structures perfectly. Thinking in midi notes, semitones, or degrees of a scale is also supported, including microtonal and alternate tuning systems, as these are easily converted to Hz based frequencies and ratios.

Lastly, more free-form and experimental approaches to dynamics and pitch are perfectly supported and represented in TimeLines too. One can arbitrarily modulate any and all parameters of a sound process over time, and custom frameworks can be built to abstract and automate workflows. Musique concrète approaches are also possible, by using SynthDefs which index into pre-recorded sound buffers and then modulating the parameters of that indexing or subsequent editing.

7.3 Flow

Even with little practice on the instrument and its techniques, I managed to achieve levels of flow and concentration above anything I had previously achieved with music software, or even other live coding environments. The main reason behind this is TimeLines' minimal mathematical syntax and abstraction of time-based patterns, which reduces the cognitive load required to translate a musical thought into an algorithm. In other words, once the musician has a clear idea of the internal structure of the music they want to create, then the process of translating that to a series of equations that combine simple modular parts is straightforward, almost like a geometrical puzzle. This ability to break down complex content into a series of simple parts prevents one from ever feeling overwhelmed by the complexity of translating their thoughts to algorithmic notation, while at the same time provides an efficient framework for building one's custom toolkit of functions that automate commonly used patterns.

On the other hand, when a certain technique or function usage has been mastered to the point where it no longer presents any challenge, a new one can easily be developed and added to one's toolkit. As most functions can be used to affect any parameter of a musical process over time, whether be it melody, harmony, rhythm, or timbre, there are always new usages for functions to be discovered and novel ways to combine them together with every other function. This provides an endless playground for musical and sonic experimentation that, no matter how skilled or experienced one is, will hardly feel limiting.

While TimeLines lacks some of the capabilities of established live coding software, such as the Tidal's intuitive triggering and manipulation of samples or SuperCollider's dynamic rewiring of the synthesis process itself, it excels in many other areas that are practically impossible to explore with any other instrument available. Even though the barrier of entry is as low as adding and multiplying signals and numbers together, the experienced user can indeed find challenges at any level of mastery; the only limit is one's imagination.

Chapter 8

Conclusions and Future Work

In conclusion, we have seen how mathematical and algorithmic notations can be combined to create a hybrid language for expressing time-evolving patterns. The implementation of such notation, based on the paradigm of Functional Reactive Programming, allows the artist to declaratively describe their thoughts and not worry about their technical implementation. By providing simple, transparent, and composable parts that are easy to learn and use with minimal syntax, the barrier of entry is lowered for the inexperienced, while at the same time the advanced and complex combinations of these primitives are available for the master to explore. By limiting one's real-time access to the inner structure of the sound process and instead providing a handful of abstracted parameters, TimeLines feels more like an instrument and less like an instrument-building workstation, providing musical exploration and inspiration rather than technological challenges. In addition, both visual and typing distractions have been reduced to a minimum, allowing the user to be immersed in the creation and manipulation musical algorithms and not on the operation of the computer that implements them.

8.1 Future Work

While TimeLines is more musically flexible and versatile than most other music software, it is far from perfect. Future study and implementations could help unlock its full potential, using it to not only control pre-defined synthesizers but also pre-recorded sound material, other music software, and external hardware instruments. In particular, MIDI, OSC, and Control Voltage (CV) support could certainly be added via SuperCollider's external interfacing capabilities, allowing TimeLines to communicate and sequence most commercially available musical equipment, from all music software to hardware sytnhs and modular synthesis setups.

In particular, more customisations to the editor could be informed by the workflow encouraged by TimeLines. Extended use of snippets could automate the scaffolding required by certain workflows, and more keyboard shortcuts can be included to aid and speed up navigation around the code, for example jumping to certain parameters of synths or selecting specific regions of the text. In addition, meta-programming the code itself using functions could be explored, as this would not only increase efficiency of typing and allow the user to manipulate code in the same high-level ways as they manipulate data, it would also allow for a scalable typing experience in which the user can always improve.

Beyond just its implementation, TimeLines is an approach to thinking about and creating music using mathematical functions of time itself. It is therefore open to be re-implemented in other languages or platforms, as long as they can provide the clean, dense, and composable syntax required for effective externalization of musical thought with minimal distractions.

Some appendix text

Bibliography

[1] Mihaly Csikszentmihalyi. *Flow: the psychology of optimal experience.* Harper Row, 2009.

[2] Henkjan Honing. From time to time: The representation of timing and tempo. *Computer Music Journal*, 25(3):50–61, 2001.

[3] Thor Magnusson. Algorithms as scores: Coding live music. *Leonardo Music Journal*, 21(21):19–23, 2011.

[4] Roger B. Dannenberg. Abstract time warping of compound events and signals. *Computer Music Journal*, 21(3):61, 1997.

[5] Karen Collins, Bill Kapralos, Holly Tessler, Chris Nash, and Alan F. Blackwell. Flow of creative interaction with digital music notations. In *The Oxford Handbook of Interactive Audio.* Oxford University Press, May 2014.

[6] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *ACM Turing Award Lectures*, 1977.

[7] J. Hughes. Why Functional Programming Matters, 1989.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. *Proceedings of the second ACM SIGPLAN international conference on Functional programming - ICFP 97*, 1997.

[9] Charles Roberts and Graham Wakefield. Tensions and techniques in live coding performance. *The Oxford Handbook of Algorithmic Music*, 2018.

[10] Alex McLean. *Artist-Programmers and Programming Languages for the Arts.* PhD thesis, Goldsmiths, University of London, 2011.

[11] Alex McLean. Making programming languages to dance to: Live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, FARM '14, page 63–70, New York, NY, USA, 2014. ACM.

[12] Andrew Sorensen and Henry Gardner. Systems level liveness with extempore. *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, 2017.

[13] Martti Nurmikari. The basics of demo programming.

[14] Alex McLean. Hacking perl in nightclubs, Aug 2004.

[15] S. Wilson, D. Cottle, and N. Collins, editors. *The SuperCollider Book*. MIT Press, Cambridge, MA, 2011.